HAL
open science

Optimisation Extensible dans un Mediateur de Données Semi-Structurées

Nicolas Travers

To cite this version:

Nicolas Travers. Optimisation Extensible dans un Mediateur de Données Semi-Structurées. Autre [cs.OH]. Université de Versailles-Saint Quentin en Yvelines, 2006. Français. NNT : . tel-00131338

HAL Id: tel-00131338

https://theses.hal.science/tel-00131338v1

Submitted on 16 Feb 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

No. : __

Université de Versailles Saint-Quentin-en-Yvelines

THÈSE

pour obtenir le grade de
docteur

discipline : **Informatique**

présentée et soutenue publiquement

par

Nicolas Travers

le 12 décembre 2006

sur le sujet

Optimisation Extensible dans un Médiateur de Données Semi-Structurées

---

JURY

Monsieur	**Patrick Valduriez**	*Rapporteur*
Monsieur	**Bernd Amann**	*Rapporteur*
Monsieur	**Georges Gardarin**	*Directeur de thèse*
Madame	**Véronique Benzaken**	*Examinatrice*
Monsieur	**Dominique Laurent**	*Examinateur*
Mademoiselle	**Tuyêt-Trâm Dang-Ngoc**	*Examinatrice*

Remerciements

Je tiens à remercier sincèrement :

Bernd Amann, Professeur de l'Université de Paris VI (LiP6) et Patrick Valduriez, Directeur de recherche de l'Institut National de recherche en Informatique et en Automatique (INRIA), pour avoir accepté de juger mon travail et pour toutes les remarques qui ont permis d'améliorer ce manuscrit.

Véronique Benzaken, Professeur CNRS au LRI à l'université de paris sud 11, et Dominique Laurent, Professeur à l'université de Cergy-Pontoise, pour avoir accepté d'examiner mon travail et de participer au jury de cette thèse.

Georges Gardarin, Professeur à l'Université de Versailles-Saint-Quentin-En-Yvelines, mon directeur de thèse qui m'a accueilli dans son équipe et m'a accordé sa confiance durant toutes ces années de thèse.

Tuyêt-Trâm Dang-Ngoc, Maître de Conférence à l'Université de Cergy-Pontoise, pour m'avoir aidé à réaliser ce travail, donné de nombreux conseils et bien sûr son amitié.

Clément Jamard pour son enthousiasme, le partage du bureau durant toutes ces années, les longues heures d'implémentations communes et les moments de rigolade.

Florin Dragan, Bogdan Butnaru et Iulian Sandu Popa pour leur bonne humeur et leur participation efficace au projet XLive.

Les joyeux thésards avec qui j'ai eu le plaisir de partager des moments inoubliables, en particulier : Parinaz Davari, Dimitre Kostadinov, David Nott, Véronika Péralta, Tao Wan, Stéphane Rousseau, Daniel Villa Monteiro et Xiaohui Xue.

Tous les membres du laboratoire PRiSM et en particulier Annick Baffert, Chantal Ducoin, Isabelle Moudener et Catherine Le Quéré pour leur gentillesse.

Les membres du projet ACI SemWeb pour les discussions, leurs soutiens et encouragement. Tous ces travaux m'ont permis de comprendre le fonctionnement de la recherche.

Tous les amis que j'ai pu rencontrer tout au long de ma scolarité, du lycée jusqu'à l'UVSQ qui m'a mené jusqu'à cette thèse : Olivier, Benoit, Jean-Charles, Arnaud, Florence, Ludovic, Erik, Vincent, Hervé, Grégoire, Cécile, Yohann, Benoit, Aurélien, Lilian, Cyrille, Julien, Vincent, Aurélie, Anne et tous les autres.

Les étudiants du cursus informatique de l'ISTY, de l'UVSQ et de Cergy Ponstoise qui m'ont fait comprendre la valeur de l'enseignement et l'importance de la pédagogie.

Et enfin du fond du cœur toute ma famille et ma belle famille pour tout l'amour qu'il me porte et pour m'avoir soutenu (et supporté) jusqu'au bout : ma femme, mes parents, Christophe, Céline, Carine, mes beaux parents, David, Anne, Serge, ma nièce Julie et mon tout jeune neveu David.

Merci à toutes et à tous à qui j'adresse toute ma gratitude.

Versailles, 6 décembre 2006
Nicolas Travers

Table des matières

1 Introduction **1**
1.1 Médiation XML . . . 2
1.2 Traitement de requêtes . . . 3
1.2.1 Traduction et Modélisation de requêtes . . . 3
1.2.2 Optimisation . . . 4
1.2.3 Évaluation d'une requête . . . 5
1.3 Contributions . . . 6
1.4 Organisation de la thèse . . . 7

2 Traitement de requêtes dans un médiateur **9**
2.1 Médiation de Données . . . 10
2.1.1 Modèles de médiation . . . 10
2.1.1.1 Intégration de données hétérogènes . . . 11
2.1.1.2 Approches de médiation . . . 11
2.1.1.3 Médiation de requêtes . . . 13
2.1.2 Architectures de médiation . . . 13
2.1.3 Systèmes de médiation . . . 14
2.1.4 Synthèse . . . 17
2.2 Traitements de requêtes XQuery . . . 17
2.2.1 Aperçu de XQuery . . . 18
2.2.2 Représentation de requêtes . . . 21
2.2.3 Canonisation de requêtes . . . 23
2.3 Optimisation Extensible . . . 27
2.3.1 Base de l'extensibilité . . . 28
2.3.2 Systèmes extensibles . . . 28
2.3.3 Synthèse . . . 32
2.4 Conclusion . . . 33

3 Tree Graph View : un modèle de représentation complet **35**
3.1 Traitement de requêtes XQuery . . . 37
3.2 Canonisation de XQuery . . . 40
3.2.1 Ordonnancement . . . 40
3.2.2 Opérations Ensemblistes . . . 42
3.2.3 Opérations Conditionnelles . . . 43
3.2.4 Séquences . . . 44
3.2.5 Fonctions . . . 45
3.2.6 XQuery Canonique . . . 46

3.3 Définition du modèle 48
3.3.1 Motifs d'Arbres 48
3.3.2 Contraintes 54
3.3.3 Hyperliens 56
3.3.4 Tree Graph Views 63
3.3.5 Conclusion sur le modèle 63
3.4 Formalisation des Tree Graph Views 64
3.4.1 Motifs d'arbre 64
3.4.1.1 Nœuds 65
3.4.1.2 Axe 65
3.4.1.3 Liens de nœuds 66
3.4.1.4 XPath 67
3.4.1.5 TreePattern 69
3.4.2 Contraintes 70
3.4.2.1 Prédicats 71
3.4.2.2 Fonctions 71
3.4.2.3 Lien de contraintes 72
3.4.2.4 Conclusion 74
3.4.3 Hyperliens 74
3.4.3.1 Hyperliens associatifs 75
3.4.3.2 Hyperliens Directionnels 76
3.4.3.3 Conclusion 80
3.4.4 Les Différents Motifs d'Arbre 80
3.4.4.1 SourceTreePattern 81
3.4.4.2 ReturnTreePattern 82
3.4.4.3 IntermediateTreePattern 83
3.4.4.4 AggregateTreePattern 83
3.4.4.5 Conclusion 84
3.4.5 Tree Graph View (TGV) 84
3.4.6 Conclusion 89
3.5 XQuery vers TGV 90
3.5.1 XQuery 91
3.5.2 SetOperator 91
3.5.3 FLWR 93
3.5.4 For 93
3.5.5 Let 94
3.5.6 Where 95
3.5.7 Return 97
3.5.8 Conclusion 98
3.6 Conclusion 99

4 Optimisation des TGV 101
4.1 Algèbre Abstraite pour TGV 103
4.1.1 Définition des opérateurs 104
4.1.2 Algorithme de décomposition 107
4.1.3 Exemple d'arbre algébrique 110
4.1.4 Conclusion 111
4.2 Annotation des TGV 112

4.2.1 Définition 112
4.2.1.1 Information 113
4.2.1.2 Liste d'éléments 113
4.2.1.3 Annotation 114
4.2.1.4 Ensemble d'annotations 114
4.2.2 Formalisation 114
4.2.2.1 Vues sur les annotations 116
4.2.3 Conclusion 118
4.3 Règles de transformation 118
4.3.1 Définition des transformations 119
4.3.1.1 Langage de règles 120
4.3.1.2 Motifs de Règles 122
4.3.1.3 Conclusion 123
4.3.2 Classification des transformations 123
4.3.2.1 Équivalence 124
4.3.2.2 Transformations logiques équivalentes 125
4.3.2.3 Transformations physiques équivalentes 125
4.3.2.4 Transformations Utilisateurs 129
4.3.3 Conclusion 131
4.4 Stratégie de recherche 131
4.4.1 Modèle de coût 132
4.4.2 Espace de Recherche 134
4.4.3 Conclusion 135
4.5 Conclusion 136

5 XLive : un système de médiation 139
5.1 Architecture de Médiation 140
5.1.1 Composants de XLive 140
5.1.2 XAlgèbre 142
5.2 Traitements de Requêtes 145
5.3 Interface Graphique et Résultats 151
5.4 Optimisation de Requêtes 153
5.5 Etudes de performances 154
5.6 Prototypes et Intégration des projets de recherche 157
5.7 Conclusion 159

6 Conclusion 161
6.1 Comparaison avec les approches existantes 162
6.1.1 Les systèmes de médiation 162
6.1.2 Les représentations de requêtes 163
6.1.3 Les optimiseurs extensibles 163
6.1.4 Synthèse 164
6.2 Contributions 164
6.2.1 Canonisation 165
6.2.2 Tree Graph Views 165
6.2.3 Optimisation extensible 165
6.2.3.1 Annotation 165
6.2.3.2 Langage de règles 166

6.2.4 XLive 166
6.3 Perspectives de recherche 167

Bibliographie **169**

Annexes **I**

A XML **I**
A.1 Description I
A.2 Espace de noms II
A.3 Métadonnées II

B XPath **V**

C XQuery **VII**
C.1 Grammaire VII
C.2 Optionalité X

D Abstract Data Types : TGV **XI**
D.1 XPath XI
D.2 TreePattern XI
D.3 Constraint XIII
D.4 Hyperlink XIV
D.5 TGV XV
D.6 Annotations XVI

E Compléments d'Algorithmes pour TreePattern **XVII**

F Cas d'usage **XIX**
F.1 Use Cases XMP XIX
F.2 Use Cases TREE XXXI
F.3 Use Cases SEQ XXXII
F.4 Use Cases R XXXIII
F.5 Use Cases SGML XXXIV
F.6 Use Cases STRING XXXV
F.7 Use Cases NS XXXVI
F.8 Use Cases PARTS XXXVII

G Propositions de modélisation **XXXIX**
G.1 Use Case STRONG XXXIX
G.2 Use Case UPDATE XL

Table des figures

2.1 Comparaison des architectures GAV et LAV 12
2.2 Architecture de médiation 14
2.3 Exemple de document XML 18
2.4 Exemple de Tree Pattern Query (TPQ) 21
2.5 Exemple de Generalized Tree Pattern (GTP) 22
2.6 Exemple de XML Access Module (XAM) 23

3.1 Cycle d'évaluation d'une requête XQuery 37
3.2 TGV de la requête du tableau 3.1 39
3.3 Ensemble de nœuds 50
3.4 Motif d'arbre appliqué à un document XML 51
3.5 Motif d'arbre simple (TreePattern) 52
3.6 Motif d'arbre source (STP) et résultat (RTP) 53
3.7 Motif d'arbre intermédiaire (ITP) et d'agrégat (ATP) 54
3.8 Hyperlien de contrainte et Hyperlien de jointure 58
3.9 Hyperlien de projection 60
3.10 Hyperlien Ensembliste 61
3.11 Hyperlien Conditionnel 62
3.12 Hiérarchie des liens 63
3.13 Diagramme d'héritage des *TreePattern*s 81

4.1 Cycle de traitement d'une requête XQuery 102
4.2 Exemple d'annotation de TGV 117
4.3 Transformation d'un TGV 119
4.4 Motifs de Règles de distributivité de la jointure 122
4.5 Motif de règle physique de permutation de jointure 126
4.6 Motif de règle physique de modification d'algorithme de jointure . . 129
4.7 Motif de règle utilisateur de changement de nom de label 130
4.8 TGV annoté par des formules de coût 133

5.1 Architecture de médiation 141
5.2 Exemple de XTuples 143
5.3 Capture d'écran de l'interface graphique de XLive 152
5.4 Performances des sources eXist et Xhive comparées au médiateur XLive, avec le cas d'usage XMP 153
5.5 Étapes d'optimisation de l'arbre algébrique 156
5.6 Graphique de performances de l'optimiseur 156
5.7 Graphique d'évolution des coefficients d'amélioration des règles . . . 157

A.1 Exemple de document XML II
A.2 Exemple de XML-Schema IV

F.1 TGV for query XMP 1 XIX
F.2 TGV for query XMP 2 XX
F.3 TGV for query XMP 3 XXI
F.4 TGV for query XMP 4 XXII
F.5 TGV for query XMP 5 XXIII
F.6 TGV for query XMP 6 XXIV
F.7 TGV for query XMP 7 XXV
F.8 TGV for query XMP 8 XXVI
F.9 TGV for query XMP 9 XXVII
F.10 TGV for query XMP 10 XXVIII
F.11 TGV for query XMP 11 XXIX
F.12 TGV for query XMP 12 XXX
F.13 TGV for query TREE 6 XXXI
F.14 TGV for query TREE 4 XXXII
F.15 TGV for query R 14 XXXIII
F.16 TGV for query SGML 8 XXXIV
F.17 TGV for query STRING 2 XXXV
F.18 TGV for query NS 7 XXXVI
F.19 TGV for query PARTS 1 XXXVII

G.1 TGV for query STRONG 4 XXXIX
G.2 TGV for query UDPATE 1 XL

Chapitre 1

Introduction

Les systèmes d'information sont de plus en plus articulés autour d'Internet ou Intranet avec des architectures orientées services (SOA). La gestion de données distribuées constitue la brique de base des systèmes d'information orientés services. Ces systèmes distribués, en marche vers le pair-à-pair, comportent des fonctions d'interrogation de complexités croissantes, facilitant l'accès et le partage des données. Les méthodes d'interrogation de données distribuées doivent tenir compte de la nature des données interrogées, dépendant du langage de manipulation spécifique à chaque source. Par exemple, les données relationnelles présentes dans de nombreux systèmes sont accessibles *via* le langage SQL ; sur d'autres systèmes, des données orientées objets sont accédées *via* OQL ou plus souvent SQL3 ; de plus en plus souvent, des données semi-structurées et textuelles sont manipulées. Ainsi, chaque source de données possède ses propres méthodes d'interrogation.

Les systèmes de médiation d'information, encore appelés EII (Entreprise Information Integrator), doivent être capable de gérer cette hétérogénéité pour fournir des vues intégrées homogène des informations dispersées. L'hétérogénéité des informations couplée à la distribution des systèmes d'information pose d'importants problèmes de localisation et d'intégration de données. En effet, un utilisateur souhaitant poser une requête sur Internet ne sera pas à même de savoir à quel système l'adresser ni même dans quel langage la formuler. Les systèmes de médiation pallient à ces besoins. Ils collectent les données et uniformisent les langages de requêtes permettant alors à l'utilisateur de n'interroger que le système de médiation pour obtenir une réponse. Cette dernière est générée à partir de sources d'information rattachées à ce que l'on appelle un médiateur de données *via* un module de traduction de requêtes et de données appelé adaptateur de source.

Nous rappelons dans cette introduction les principes de la médiation, ainsi que les problématiques qui lui sont attachées. Nous introduisons ensuite le processus de traitement de requêtes comprenant les phases de représentation, d'optimisation et d'évaluation. Enfin, nous décrivons notre approche de résolution de cette problématique dans le contexte semi-structuré (XML) qui est au cœur de cette thèse. Nous

soulignons notamment les apports de cette thèse et présentons les chapitres essentiels.

1.1 Médiation XML

Internet est un vaste entrepôt de données réparties irrégulièrement dans le monde. Les systèmes d'information d'entreprise autour d'Intranets peuvent aussi être vus comme des entrepôts plus ciblés. Le principal problème lié à la recherche d'information dans ces systèmes est de pouvoir trouver, rassembler et produire un résultat compréhensible aussi complet que possible pour répondre à une requête. Une des solutions envisageables pour répondre aux problèmes liés à la distribution et l'hétérogénéité des sources d'information est d'utiliser un médiateur de données. Un médiateur centralise un ensemble de sources d'information, communique avec elles, traite les données qu'il peut en retirer, et produit un résultat global pour l'utilisateur. De plus, le médiateur qui possède son langage de requête (appelé langage pivot), doit pouvoir dialoguer avec chaque source de données selon son langage spécifique. Aussi, le traitement de requêtes peut être long ; des transformations et des optimisations de requêtes permettent souvent de réduire le temps de traitement et de faciliter le passage à l'échelle.

Le traitement des requêtes utilisateurs dans un système de médiation pose un grand nombre de problèmes qu'il n'est pas aisé de résoudre. En effet, un médiateur doit être capable de gérer un grand nombre de sources de données *via* des adaptateurs, l'architecture d'ensemble permettant de produire une réponse uniforme à l'utilisateur. Les traitements effectués par le médiateur doivent être totalement transparent à l'utilisateur, le médiateur doit lui apparaître comme une unique source de données homogènes. De plus, les sources de données ne supportent que rarement le langage de requête pivot du médiateur. Les informations gérées ne sont pas forcément identiques et les sources n'ont pas toutes les mêmes capacités de fonctionnement. Ainsi, le système de médiation doit être capable de pallier à ces problèmes qui sont liés à son contexte opératoire. Plus particulièrement, le médiateur doit être capable de comprendre les requêtes de l'utilisateur, reconnaître les sources d'information capable d'y répondre, récupérer les informations dans le format pivot et produire un résultat unique.

De nombreux langages de requêtes ont été conçus pour répondre aux différentes structures de données existantes. Les données relationnelles et le langage SQL ont été les standards les plus usités par le passé, puis les données Orientés Objets et les langages OQL et SQL3 ont répondus à d'autres demandes pour les systèmes d'information. Enfin, les données semi-structurées **XML**[Bray *et al.* 1997] et son langage de requête **XQuery**[W3C 2006a] ont vu le jour à la fin des années 90 en réponse à l'essor des technologies provenant d'Internet. Dans le contexte XML, les traitements sont souvent lourds et les données volumineuses ; il est donc essentiel d'utiliser des techniques d'optimisation efficaces pour améliorer les performances du médiateur. Des techniques de transformation et de réécriture de requêtes, d'indexation de don-

nées, de copies (ou cache) des résultats de requêtes sont utilisables pour remédier à ces problèmes de performances.

Dans le cadre de notre travail, nous nous concentrons sur l'optimisation de requêtes. Ce processus complexe repose sur un ensemble de principes que nous développons tout au long de cette thèse. Ainsi, un système de médiation «tout-XML» permet d'intégrer un grand nombre de sources de données hétérogènes en XML. De plus, il doit être capable de dialoguer avec chacune d'elles dans leurs langages spécifiques. Enfin, le traitement des requêtes doit pouvoir être optimisé afin d'assurer le passage à l'échelle du médiateur. Dans cette thèse, nous développons les concepts et les techniques de médiation liées aux données semi-structurées, en particulier basés sur le langage de requêtes **XQuery**. Les résultats ont été appliqués à la construction du système de médiation **XLive** [Dang-Ngoc *et al.* 2005], système de médiation *open-source* développé au laboratoire PRiSM et utilisé avec succès dans plusieurs projets nationaux et européens. Une bonne partie de nos résultats ont été mis en œuvre dans ce système en cours de mise en logiciel libre.

1.2 Traitement de requêtes

Le traitement de requêtes vise à traduire une requête utilisateur en une structure compréhensible par le médiateur, lui permettant de produire une réponse adéquate. Ce processus de traduction est complexe, et repose sur cette structure qui doit être conçue pour faciliter les différentes étapes du traitement de requêtes. Ces étapes décomposent la requête en un plan d'exécution composé de sous-requêtes mono-source et d'une requête de synthèse. Nous distinguons trois composants principaux dans le traitement de requêtes : *analyse*, *optimisation* et *évaluation*. L'analyse permet de construire un modèle représentant la requête de manière compréhensible pouvant ensuite être manipulé. L'optimisation de requêtes manipule, quant à elle, le modèle de représentation pour en améliorer l'évaluation et produire un plan optimisé. L'évaluation de requête permet, à partir du plan construit, de produire un résultat intégrant les données extraites des sources. Ainsi, la conception d'un modèle de représentation de requêtes adéquate à chacune de ses étapes est indispensable au bon fonctionnement du traitement de requêtes.

1.2.1 Traduction et Modélisation de requêtes

La traduction du langage de requêtes doit être capable de reconnaître la grammaire et la syntaxe de ce langage. Ses éléments caractéristiques doivent être identifiés ; une structure adéquate permet de préserver les informations qu'une requête contient.

La modélisation de requêtes permet de construire une structure aisément manipulable pour faciliter le traitement de la requête à l'intérieur du gestionnaire de données. Pour bien modéliser une requête :

1. chaque caractéristique du langage de requête doit pouvoir être traduite et correspondre à une structure particulière dans le modèle ;
2. la structure retenue doit être flexible pour faciliter les transformations de requête ; ainsi, les informations doivent être facilement modifiables et les liens entre les instances interchangeables ; bien sûr, ces modifications doivent respecter le sens de la requête, donc conserver le résultat qu'elle doit produire ;
3. une représentation visuelle graphique de la requête doit être possible afin de faciliter la compréhension des opérations en mettant en avant les caractéristiques essentielles du langage de requêtes.

Le modèle de représentation de requêtes est le point central du traitement de requêtes. Il permet de faciliter chacune des étapes du traitement de requêtes effectuées dans le système.

1.2.2 Optimisation

L'optimisation de requêtes consiste en un ensemble de transformations que l'on applique sur une instance du modèle de représentation pour en améliorer l'efficacité (temps d'évaluation, espace mémoire, processeur, coût de communication...). L'optimiseur repose donc sur le modèle de représentation qu'il manipule. Un optimiseur se décompose en différents modules dont le module de *transformation*, le module de *calcul* et le module d'application de *la stratégie de recherche*.

Les *règles de transformation* donnent les conditions pour lesquelles une transformation peut être appliquée et les modifications qui résultent de cette application. Ainsi, si les caractéristiques d'une requête répondent aux pré-requis d'une règle de transformation, alors, celle-ci peut être transformée selon les actions de transformation données par la règle. Bien sûr, ces règles ne doivent pas modifier le résultat de la requête, quelque soit la transformation. Il faut donc que les conditions d'application de la règle garantissent l'équivalence des résultats avant et après la transformation de la requête.

Le *modèle de coût* permet de donner une valeur quantitative à une requête modélisée. Cette valeur théorique permet d'estimer le temps d'évaluation de la requête en temps processeur et en temps entrées-sorties. Le modèle de coût permet à l'optimiseur de prédire un temps pour chaque plan possible et ainsi de choisir le meilleur. Il repose sur un ensemble de formules donnant une valeur à chaque opération dérivée du modèle de requête, l'ensemble permettant de donner une valeur théorique au plan d'exécution

La *stratégie de recherche* relie les règles de transformation au modèle de coût pour permettre à l'optimiseur de produire le plan le plus optimal possible. Le principe de la stratégie de recherche est de réduire l'ensemble des solutions possibles par transformations. Cet ensemble de solutions possibles est appelé *espace de recherche*. Si l'optimiseur applique l'ensemble de toutes les transformations possibles sur une requête, l'espace de recherche est *complet*, contenant toutes les solutions possibles.

La solution optimale est alors la requête dont le coût d'évaluation donné par le modèle de coût est minimal. Malheureusement, la génération de l'espace de recherche complet peut prendre un temps important (en particulier dans le cas de requêtes de grande taille). C'est pour cela qu'une stratégie de recherche est nécessaire pour réduire l'espace de recherche. La stratégie de recherche consiste à orienter la sélection des règles de transformation afin d'obtenir un plan proche de l'optimal en un temps raisonnable.

Un *optimiseur extensible* est un optimiseur dont la capacité d'optimisation peut être modifiée au cours du temps. En effet, un optimiseur classique comporte des techniques d'amélioration fixe (suite de transformations précises), l'optimiseur reste immuable. Les connaissances d'un optimiseur extensible peuvent être augmentées pour lui permettre une meilleure optimisation. Ainsi, des règles de transformation peuvent être ajoutées ; d'autres peuvent être changées ; la stratégie de recherche peut être modifiée pour que l'espace de recherche soit parcouru plus efficacement. En résumé, le processus d'optimisation intègre donc un ensemble de techniques complexes qu'il faut automatiser pour permettre de générer le plan de requête que le médiateur pourra alors évaluer en un temps optimal. A noter qu'un plan peut être conservé et évalué de multiples fois, ce qui multiplie l'intérêt d'une bonne optimisation.

Il est donc nécessaire de concevoir un optimiseur où chaque module repose sur le modèle de représentation de requêtes. Chacun de ces modules demande une étude précise pour permettre de produire un plan de requêtes le plus efficace possible.

1.2.3 Évaluation d'une requête

Une fois le plan d'exécution élaboré par l'optimiseur, la requête modélisée peut alors être évaluée pour obtenir le résultat. Le plan se décompose en un arbre d'opérations, chaque opération permettant de traiter des données pour produire un résultat particulier. Les opérations reposent sur une *algèbre*. Une requête à évaluer est représentée par un arbre d'opérations de cette algèbre communément nommée un *plan algébrique*.

Certains systèmes de gestion de données utilisent un modèle de requête commun pour les étapes d'optimisation et d'évaluation de requêtes, tandis que d'autres systèmes utilisent des modèles différents pour chaque étape. Nous parlons alors de *représentation logique* pour le modèle de requêtes et de *plan physique* pour l'algèbre d'évaluation.

Dans le cadre de la médiation, des opérateurs sont ajoutés à l'algèbre classique pour assurer l'intégration de données. En effet, ces opérations constituent les intermédiaires entre l'arbre algébrique global et les opérations sur les sources de données distantes, en quelques sortes elles symbolisent la glue entre le plan global du médiateur et les opérations locales. De plus, une source de données n'a pas forcément toutes les capacités de traitement nécessaires pour une requête soumise au médiateur ; dans ce cas, ce dernier doit être capable de pallier à ces absences de fonctionnalités. De

nouvelles opérations doivent alors être intégrées à l'arbre algébrique pour effectuer les opérations non reconnues par la source de données. L'évaluation de requêtes est un processus bien structuré permettant de produire efficacement le résultat. Son fonctionnement devient plus complexe dans un contexte de médiation. En effet, la localisation des données distantes, la fusion des données dans le médiateur, la délégation des traitements aux sources sont des tâches supplémentaires qu'un médiateur doit être capable de résoudre pour fonctionner convenablement.

L'évaluation de requêtes dans le contexte de la médiation demande des traitements supplémentaires qu'il faut intégrer pour permettre une évaluation complète. Chaque source capable de répondre à une requête doit être intégrée au plan d'exécution.

1.3 Contributions

Dans cette thèse, nous proposons un processus de modélisation, optimisation et évaluation de requêtes complet et efficace dans un cadre d'évaluation de requêtes hétérogènes semi-structurées répondant aux exigences de la médiation et du traitement de requêtes. Le système de médiation «tout-XML» **XLive**[Dang-Ngoc *et al.* 2005], basé intégralement sur les technologies XML, intègre ce processus pour évaluer des requêtes XQuery et produire des résultats en XML. Notre approche aborde toutes les problématiques liées à la médiation et l'optimisation de requêtes dans le contexte XML/XQuery.

Afin de recouvrir la plupart des caractéristiques du langage XQuery, nous proposons des règles de transformation de requêtes pour faciliter le travail de traduction dans notre modèle de représentation. La requête transformée par ces règles donne alors une structure particulière appelée *requête canonique*. Nos résultats complètent ceux de [Chen 2004] en introduisant une forme canonique de requête couvrant une plus large part de XQuery.

Nous proposons d'autre part un modèle de représentation de requêtes, basé sur les principes proposés par [Chen 2004] et [Amer-Yahia *et al.* 2001]. Ce modèle de représentation permet de modéliser toutes les requêtes XQuery non typées : les **Tree Graph Views** (TGV). Le langage XQuery est complexe et comporte des caractéristiques bien particulières. Nous les exploitons pour produire un modèle logique, intuitif et aisément manipulable. Notre modèle prétend être le plus complet possible, recouvrant une grande partie de la grammaire du langage XQuery. Une structure physique d'évaluation dérivée de ce modèle donne une base permettant de produire un plan d'exécution pour l'évaluation. Le modèle a aussi l'intérêt de permettre une représentation graphique claire des requêtes.

Une technique d'optimisation est alors proposée pour manipuler notre modèle de représentation et produire un plan optimal. L'extensibilité de l'optimiseur se caractérise par un langage de définition de règles capable d'intégrer de nouvelles règles de transformation dans l'optimiseur. Afin de réduire l'espace de recherche généré

par l'optimiseur, nous proposons une stratégie de recherche manipulant les règles de transformation créées. Le modèle de coût, qui mesure le temps théorique de chaque requête et permet à l'optimiseur de choisir le plan optimal, est ensuite introduit lui-même entrant dans le cadre d'une autre thèse [Liu to appear]. L'intérêt de nos techniques d'optimisation est validé sur un ensemble de bancs d'essai [Dragan et Gardarin 2005] («*benchmark*») et de cas d'usage («*use-cases*») du [W3C 2006b].

Le système de médiation XLive [Dang-Ngoc *et al.* 2005] est lui-même une contribution de cette thèse. Le système de médiation XLive intègre *via* des adaptateurs des sources de données variées (SQL, XQuery, Fichiers XML, LDAP ...). Le traitement des requêtes est basé sur le modèle de requêtes proposé : les TGV. Le langage de règles, l'optimiseur et l'évaluation des requêtes formalisés dans cette thèse sont au cœur du système de médiation *XLive*. Le logiciel XLive est mis en logiciel libre et a été utilisé dans plus plusieurs projets : projets européens IST *WebSI* [WebSI 2004], Satine [Satine 2004], le projet ACI SemWeb [SemWeb 2004] et le projet ANR PADAWAN [Padawan 2005].

1.4 Organisation de la thèse

Après la présente introduction, nous procédons à une analyse de l'ensemble des travaux en relation avec notre sujet dans le *chapitre 2*. Ce chapitre présente les principes de médiation, ainsi que quelques exemples de systèmes qui ont orienté notre réflexion. Nous nous penchons ensuite sur différents langages de requêtes pour des données XML. Nous nous focalisons plus particulièrement sur les caractéristiques du langage de requêtes XQuery, et les moyens de le représenter. Pour finir, nous portons notre réflexion sur les systèmes d'optimisations extensibles et leurs techniques de réduction de l'espace de recherche qui ont inspiré les choix de nos travaux.

Ensuite, nous présentons dans le *chapitre 3* notre modèle de représentation de requêtes XQuery : les **Tree Graph Views**. Le chapitre est décomposé en quatre sections distinctes. La première partie définit les règles de canonisation des requêtes XQuery pour obtenir une requête canonique, préparant l'étape de traduction des requêtes. La seconde partie définit le modèle de représentation de requête : chaque élément du modèle est associé à une représentation graphique donnant une approche intuitive de visualisation de la requête. La troisième partie de ce chapitre donne la formalisation du modèle sous forme de *types abstraits de données* permettant la définition exacte du fonctionnement de notre modèle. Enfin, la quatrième partie donne la procédure de traduction d'une requête XQuery canonique vers le modèle *TGV*. Chaque algorithme reprend les caractéristiques du langage pour les faire correspondre à celles spécifiées dans notre modèle. Pour clore ce chapitre, un exemple illustre les étapes de transformation d'une requête XQuery vers les *TGV*.

Dans le *chapitre 4*, nous proposons un *optimiseur extensible*. Une algèbre abstraite d'évaluation des *TGV* est proposée, permettant de définir la méthode d'évaluation des requêtes traduites dans notre modèle. Puis, une base d'annotation des *TGV*

permet d'introduire des informations supplémentaires à notre modèle, permettant alors de mesurer le coût des plans. Ensuite, un langage de définition de règles de transformation est introduit pour définir de nouvelles règles d'optimisation, base de l'extensibilité de notre système. Une représentation graphique de ces règles de transformation est proposée pour en faciliter la compréhension. Enfin, une stratégie de recherche manipulant les règles de transformation intégrées dans l'optimiseur est décrite. Elle permet de réduire l'espace de recherche généré par l'optimiseur.

L'architecture du système **XLive** est présentée au *chapitre 5*. Les différents modules et leurs interactions y sont détaillés. Une étude qualitative du modèle de représentation de requêtes TGV est proposée, et une étude quantitative présente les performances de l'optimiseur de requêtes à l'aide d'un «*benchmark*» très simple, spécifique à la médiation de requêtes.

Nous concluons cette thèse dans le *chapitre 6* en résumant chacune des contributions de notre travail de recherche. Un positionnement comparatif par rapport aux travaux existants est aussi proposé. Nous terminons en introduisant quelques perspectives de recherche.

Chapitre 2

Traitement de requêtes dans un médiateur

Pour permettre l'interrogation de sources de données hétérogènes et distribuées, une architecture de médiation [Wiederhold 1992] est généralement utilisée. Une telle architecture est composée d'un *médiateur* et d'*adaptateurs*.

Les sources de données qu'un médiateur fédère sont interrogeables avec leur propre langage de requête basé sur leur propre modèle de données. Ainsi, un adaptateur doit pouvoir connaître ce langage, traduire les requêtes en langage pivot en requêtes dans le langage reconnu par la source, et aussi faire correspondre le modèle de données de la source à celui utilisé par le médiateur. Une des tâches du médiateur avec ses adaptateurs est donc de communiquer avec les sources dans leur propre langage.

Le traitement des requêtes dans un système de médiation dépend de la structure des données ainsi que du langage utilisé pour les interroger. Ce processus se décompose en trois étapes principales :

1. *analyse* ;
2. *optimisation* ;
3. *évaluation*.

Le module d'analyse de requêtes permet de transformer celle-ci en un modèle de représentation, utilisé à l'intérieur du système de médiation. Cette représentation permet aux autres modules de la transformer pour achever le processus de traitement de requêtes.

Le module d'optimisation contient un ensemble de composants permettant de manipuler le modèle de représentation de requêtes et de le compléter par des annotations. Les transformations appliquées tendent à rendre l'évaluation de la requête la plus optimale possible grâce à un ensemble de techniques d'optimisation. L'extensibilité d'un optimiseur est définie par sa capacité à ajouter des règles d'optimisation.

L'évaluateur de requêtes dans un médiateur doit tenir compte de chacune des sources disponibles, et de la manière de les combiner. Le résultat final est produit à partir des données récupérées depuis les sources puis intégrées.

Dans ce chapitre nous abordons le problème d'optimisation de requêtes sur des données semi-structurées XML dans un contexte de médiation. Pour commencer, nous détaillons les différents systèmes de médiation de données XML (section 2.1) ce qui nous permet de voir les techniques abordées pour répondre aux problèmes de médiation. Puis les techniques de représentation et de modification des requêtes sont étudiées dans la section 2.2. Enfin, nous terminons par une étude comparative de différentes techniques d'optimisation extensible (section 2.3) détaillant les langages de règles proposés et les méthodes de réduction de l'espace de recherche.

2.1 Médiation de Données

Aujourd'hui, les sources de données sont multiples et disséminées sur des serveurs différents gérés par des applications variées. Celles-ci sont parfois disponibles *via* Internet. Chacune des sources peut être interrogée indépendamment afin de récupérer les informations requises. Toutefois, l'interrogation de sources de données distantes demande beaucoup de traitements [Sheth et Larson 1990].

Lorsque plusieurs sources peuvent répondre à une requête, il faut :

1. Poser cette question à chacune d'elles ;
2. Récupérer les résultats, et les regrouper afin de les analyser ;
3. Construire la réponse souhaitée.

La quantité de données à traiter devient alors importante et demande une étape d'optimisation. Il faut aussi communiquer avec chaque source. Chacune d'elles possède un protocole de communication, un langage d'interrogation, une structure de résultats, ce qui pose un problème de compatibilité afin d'unifier les sources pour produire un résultat unique et uniforme.

La médiation de données propose une solution à ces besoins. En effet, elle permet d'interroger de vastes sources de données virtuelles intégrant tout ou partie des données de chacune des sources, tout en utilisant des protocoles d'échange adaptés à chaque source. Les systèmes de médiation de données cherchent à retrouver rapidement les informations disséminées en facilitant les applications par le biais de requêtes unifiées.

2.1.1 Modèles de médiation

L'objectif d'un médiateur est d'intégrer un grand nombre de sources hétérogènes disséminées. Il doit pouvoir intégrer les données pour répondre aux requêtes des

utilisateurs. Une requête soumise par l'utilisateur est traitée par le médiateur. Elle produit, grâce à un ensemble de transformations définies dans le médiateur, un ensemble de sous-requêtes qui sont alors transmises aux adaptateurs. Ces derniers communiquent ensuite avec les sources. Les résultats sont intégrés par le médiateur qui créé le résultat final correspondant à la demande de l'utilisateur.

Les traitements de médiation posent certains problèmes de conception. En effet, un médiateur doit traiter les points importants du cycle de vie d'une requête dans un contexte hétérogène et distribué. Il doit effectuer la transformation de la requête soumise par l'utilisateur, gérer les informations nécessaires à son exécution, générer les protocoles adéquats pour chaque source interrogée, réécrire chaque sous-requête - produit de la transformation de l'original - en un langage compréhensible par la source ciblée, reconstruire le résultat de la source pour qu'il puisse correspondre à la structure définie dans le médiateur, et enfin produire le résultat requis par l'utilisateur.

2.1.1.1 Intégration de données hétérogènes

L'intégration des données hétérogènes consiste dans un premier temps à spécifier les entités à intégrer et dans un deuxième temps à intégrer les données elles-mêmes. Des anomalies fréquentes dans l'intégration des données peuvent alors survenir :
- *Redondance des données* ; lorsque deux sources contiennent des données identiques, leurs intégrations provoquent une redondance au niveau du médiateur ;
- *Correspondance des noms* ; deux entités peuvent avoir la même définition sémantique, mais une structure distincte, cela peut faire disparaître des informations lors de l'interrogation ;
- *Similitude des noms* ; à l'opposé, deux entités peuvent avoir la même structure mais des définitions sémantiques distinctes, cela provoque des incohérences de résultat ;

L'intégration de plusieurs sources pose alors les problèmes suivants :
- comment aider l'utilisateur à formuler des requêtes valides ? comment lui présenter des méta-données adaptées à la médiation ;
- comment structurer les méta-données pour aider l'évaluation des requêtes ? comment identifier les sources nécessaires à l'exécution des requêtes ?

2.1.1.2 Approches de médiation

Afin d'intégrer des données distribuées au sein d'un médiateur, on distingue deux approches [Levy 1999] pour établir la correspondance entre un schéma global et les schémas des sources de données (voir figure 2.1). D'une part, l'approche *local-as-view* (LAV) où les vues locales participent à la création de la vue globale. D'autre part, l'approche *global-as-view* (GAV) où le schéma global est défini comme une vue sur les schémas locaux.

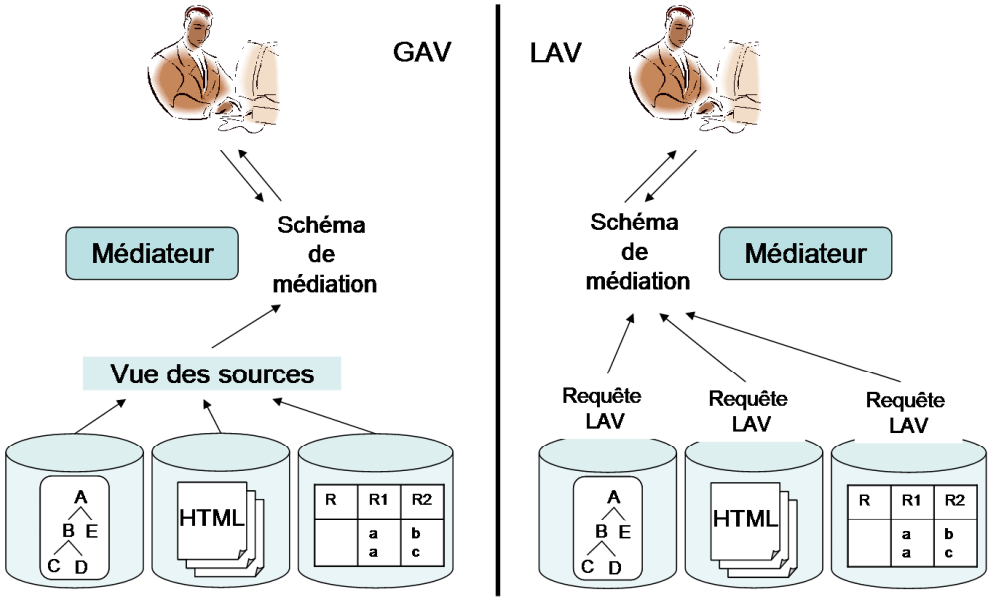


FIG. 2.1 – Comparaison des architectures GAV et LAV

Approche LAV. Dans cette approche, le schéma des sources est défini en fonction du schéma global de médiation. Les sources créent des requêtes décrivant leur données en fonction du schéma de médiation. Cette approche permet d'intégrer ou de retirer facilement une source. En effet, chaque source est définie de manière indépendante des autres sources. Sa définition n'influe donc pas sur la définition du schéma global de médiation. La contrepartie de cette flexibilité est le coût de l'évaluation d'une requête sur le schéma de médiation puisqu'il est nécessaire de réécrire chacune des requêtes des sources selon un processus complexe d'inversion.

Approche GAV. Le schéma de médiation est défini en fonction des schémas sources. Le médiateur doit créer les requêtes permettant d'interroger chacune des sources pour obtenir les données du schéma de médiation. Contrairement à l'approche *LAV*, l'évaluation d'une requête sur le schéma de médiation consiste en une simple réécriture de la requête en fonction de chacune des sources. Par contre, cette approche demande une réécriture de la vue globale à chaque suppression, ajout ou modification d'une source.

Les architectures les plus souvent utilisées sont des architectures GAV (TSIMMIS, GARLIC, DISCO, Xperanto, STRUDEL, LiquidData). Très peu utilisent l'approche LAV (Information Manifold, AGORA, LeSelect).

2.1.1.3 Médiation de requêtes

Un médiateur présente des vues intégrées des sources de données. Lorsqu'un utilisateur interroge le médiateur, la requête est posée indépendamment de la localisation des différentes données intervenant pour calculer le résultat. Par rapport à l'interrogation d'une source unique, la médiation demande trois étapes supplémentaires :

- *Décomposition de requêtes* ; il s'agit d'identifier puis d'isoler les différentes parties de la requête correspondant à une localisation des données. La phase d'identification repose sur une connaissance des sources via les méta-données permettant de localiser les sources ciblées. La phase d'isolation crée des requêtes spécifiques pour les sources à partir de ces sous-parties.

- *Recomposition des résultats* ; Les résultats obtenus à partir des sous-requêtes envoyées aux sources sont recomposés pour obtenir le résultat de la requête initiale. Dans ce processus, le moteur d'évaluation peut effectuer des opérations additionnelles sur les résultats dans le cas de dépendances entre des sous-requêtes (i.e., jointure, agrégat, union) ou du manque de capacité de traitement au niveau de la source (i.e., fonction de recherche de valeur, agrégat).

- *Optimisation du plan global* ; Le médiateur n'a pas de vision globale des coûts de traitement des sous-requêtes par les sources. L'optimisation du plan défini par le médiateur doit s'appuyer sur un modèle de coût intégrant les différentes stratégies d'évaluation proposées par les sources (i.e. stockage, indexation) ainsi que celles du médiateur. L'optimisation consiste à composer le plan en fonction de l'ensemble de ces informations.

2.1.2 Architectures de médiation

Les architectures des bases de données fédérées ont progressivement convergé vers une vue unifiée proposée par DARPA I3 [Patil *et al.* 1992] et Gio Wiederhold [Wiederhold 1992] représentée dans la figure 2.2.

Cette architecture se compose de trois niveaux :

1. *Le niveau source* : comporte les différentes sources de données. Les sources de données communique avec le médiateur à l'aide d'*adaptateur* (*wrapper*) publiant de manière homogène les données de la source. L'*adaptateur* est chargé de *(i)* traduire une requête exprimée dans le langage du médiateur en une requête exprimée dans le langage de la source, *(ii)* faire évaluer la requête par la source et *(iii)* renvoyer les résultats au médiateur.
2. *Le niveau médiateur* : comporte un ou plusieurs médiateurs permettant d'intégrer les données transmises par les *adaptateurs* des sources. Il s'occupe de faire l'interface entre une requête utilisateur et les sources évaluant la requête suivant le modèle *décomposition, recomposition, optimisation* cité précédem-

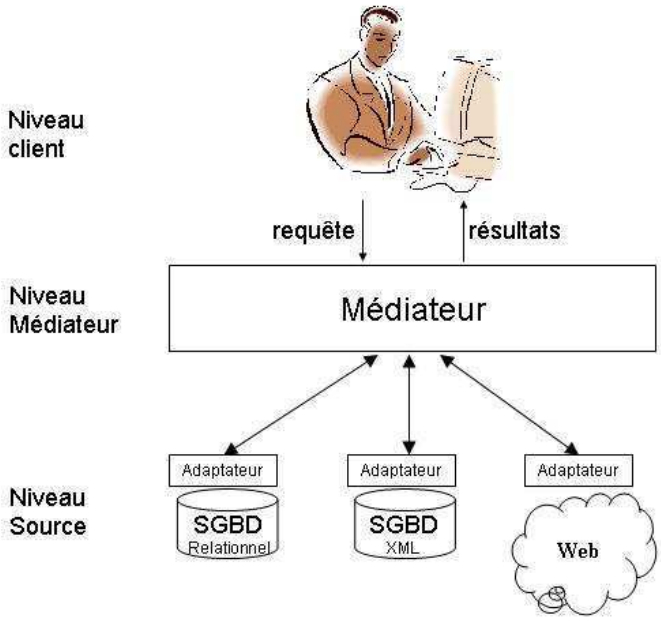


FIG. 2.2 – Architecture de médiation

ment. Il présente les données de manière homogène et centralisée à la couche supérieure, résolvant le problème d'hétérogénéité et de distribution des données.

3. *Le niveau client* : comporte les application clientes pour accéder aux médiateurs (i.e. navigateur, interface graphique, API cliente).

L'ajout d'une nouvelle source à un système de médiation entraîne la définition d'un nouvel *adaptateur* pour la rendre accessible [Cluet *et al.* 1998].

2.1.3 Systèmes de médiation

La plupart des systèmes de médiation sont basés sur l'architecture DARPA I3. Dans les années 80, les premières architectures de médiation (MULTIBASE [Landers et Rosenberg 1982] MERMAID [Brill *et al.* 1984] Information Manifold [Kirk *et al.* 1995]) s'appuyaient sur des SGBD relationnels. Ensuite, avec l'avènement des systèmes de base de données orientés objets de nouvelles architectures de médiations sont apparu (IRO-DB [Gardarin *et al.* 1995], PEGASUS [Shan *et al.* 1995], GARLIC [Haas *et al.* 1997], DISCO [Tomasic *et al.* 1996]). A la fin des années 90, avec l'apparition des systèmes de gestion de données semi-structurées, une nouvelle génération de médiateur a vu le jour (TSIMMIS [Chawathe *et al.* 1994], YAT [Cluet *et al.* 1998], AGORA [Manolescu *et al.* 2001], Xperanto [Fan *et al.* 2002], STRUDEL [Fernandez *et al.* 1998], STYX [Fundulaki *et al.* 2002][Amann *et al.* 2002], PIAZZA [Halevy *et al.* 2003a], SOMEWHERE [Adjiman *et al.* 2004]). Dans la suite nous approfondissons les architecture de médiation prenant en compte les données semi-structurés.

TSIMMIS TSIMMIS (*The Stanford IBM Manager of Multiple Information Sources*) [Chawathe *et al.* 1994] [Garcia-Molina *et al.*] est un projet permettant de développer des outils qui facilitent l'intégration de sources d'information hétérogènes comportant des données structurées et semi-structurées. Un des objectifs de TSIMMIS est d'intégrer des sources qui sont très hétérogènes, qui peuvent être peu structurées et qui sont susceptibles d'évoluer rapidement. TSIMMIS a introduit le formalisme OEM [Papakonstantinou *et al.* 1995] comme modèle de donnée interne et a adopté un langage de requête dérivé d'OQL : OEM-QL. Afin de pouvoir utiliser une source de données purement semi-structurée, un SGBD semi-structuré natif LORE, basé sur OEM, a aussi été développé.

GARLIC Le projet GARLIC [Haas *et al.* 1997] est un projet semblable développé au centre de recherche IBM Almaden. Son objectif est l'intégration de diverses sources multimédia (image, vidéo, texte, objets médicaux, cartes géographiques). Le modèle de données commun de GARLIC est orienté objet, et dispose de son propre langage de requête étendu du SQL : GQL (*Garlic Query Langage*). La différence entre GARLIC et TSIMMIS est dans la prise en compte des capacités des sources. Le médiateur TSIMMIS assume que les adaptateurs qu'il interroge sont autonomes [Li *et al.* 1998] rendant la décomposition et l'optimisation des requêtes plus simple à réaliser. En contrepartie la création des adaptateurs est plus lourde. Avec GARLIC, le médiateur prend en considération les capacités des sources rendant les algorithmes de traitement plus complexe, et un travail supplémentaire pour le médiateur mais permettant une optimisation plus importante.

LiquidData LiquidData [Carey 2004] est une architecture de médiation tout XML. Les applications clientes et les adaptateurs accèdent au médiateur de LiquidData en envoyant des requêtes en XQuery. Les résultats renvoyés sont en XML. Le serveur LiquidData transforme la requête XQuery en un plan logique optimisé, qu'il transforme ensuite en un plan physique. Ce plan physique est ensuite exécuté sur les différentes sources via leurs adaptateurs, et recompose les résultats. Les plans physiques sont optimisés à l'aide de règles simples : ordonnancement des sources, algorithmes de jointures pour permettre une évaluation la plus rapide possible

AGORA / LeSelect AGORA [Manolescu *et al.* 2001] [Manolescu *et al.* 2000] est une des seules architectures de médiation à utiliser l'approche LAV. Pour cela il s'appuie sur le médiateur LeSelect [Caravel 1998] (médiateur de données relationnel) dont le modéle de données a été étendu pour le semi-structuré. L'objectif d'AGORA est de supporter les requêtes et l'intégration de sources relationnelles et semi-structurées. Le schéma global de AGORA est une DTD XML. Les sources relationnelles et XML sont décrites comme vue sur ce schéma global. Un schéma relationnel générique est utilisé comme interface entre ce schéma et les sources. Pour chaque type de nœud (élément, attribut, texte) une table relationnelle est associée dans ce schéma. De cette façon les requêtes XQuery sont transformées en requête relationnelle et les résultats sous forme de tuples ensuite retransformés en XML.

XPeranto XPeranto [Fan *et al.* 2002] [Carey *et al.* 2000a] [Carey *et al.* 2000b] tout comme AGORA transforme un langage de requête orienté XML (XML-QL) en SQL. Pour cela il passe par un langage intermédiaire XQGM (*XML Query Graph Model*) - un langage neutre de représentation intermédiaire de requête sur XML - un composant XML tagger permet ensuite de convertir les tuples en documents XML.

DISCO Le médiateur **DISCO** (*Distributed Information Search COmponent*) [Tomasic *et al.* 1996] [Kapitskaia *et al.* 1997] [Tomasic *et al.* 1997] [Naacke 1999] a été bâti sur la notion d'intégration de capacités décrie par *Information Manifold* [Levy *et al.* 1996] et nécessitant un éditeur d'adaptateur afin de décrire les capacités d'une source de données. Cette description repose sur un langage basé sur un ensemble d'opérateurs logiques (relationnels) tel que *select, project* et *scan.*

Piazza / SOMEWHERE Les systèmes d'intégration Piazza [Halevy *et al.* 2003a] [Halevy *et al.* 2003b] et SOMEWHERE [Adjiman *et al.* 2004][Rousset *et al.* 2006] sont des systèmes de médiation *pair à pair.* Il n'existe pas de schéma de médiation global, mais chaque pair définit des *mappings* vers d'autres pairs.

Le système Piazza a d'abord été défini pour un modèle de données relationnel. Il a ensuite été étendu pour intégrer des données XML et RDF (*Resource Description Framework* [W3C 2004b]). Les pairs RDF qui sont décrits sous une ontologie *OWL* [W3C 2004a] possèdent déjà le langage de *mapping* entre deux ontologies, tandis que les pairs XML décris en XML-schema nécessitent de définir un *mapping* entre eux exprimé en XQuery. Les mappings sont directionnels d'un nœud source S vers un nœud cible C. Suivant que la requête est posée sur S ou C, la requête doit être composée dans le cas d'une requête sur C (approche GAV), ou bien reformulée dans le cas d'une requête sur S en utilisant le mapping inverse (approche LAV). Ce système présente des méthodes pour récrire les requêtes XQuery en utilisant les mappings inverse.

Dans le système SOMEWHERE, toutes les données sont décrites en un *OWL-Lite.* OWL PL (*Propositional Logic*) est le fragment de *OWL DL* réduit à l'expressivité de la disjonction, la conjonction et la négation, pour construire les descripteurs de classes décrivant le schéma des données. Chaque pair défini :
- une ontologie propre avec des classes intentionnelles correspondant à la description des données qu'il possède ;
- des classes extensionnelles, correspondant à la description de vues des données qu'il partage ;
- des mappings entre son ontologie et les ontologies de ses voisins, correspondant à la relation entre descripteurs de classes de deux pairs.

Chaque requête posée à un pair interroge les classes intentionnelles définies par celui-ci. En utilisant les classes extensionnelles, la requête est envoyée aux pairs qui lui sont reliés, puis récursivement sur tout le réseau. Cela garantit d'atteindre toutes les réponses possibles sur le réseau.

Par rapport à SOMEWHERE, Piazza a besoin d'une connaissance globale des mappings pour évaluer les requêtes. Ce qui induit qu'il existe une partie centralisée sur la connaissance des mappings.

2.1.4 Synthèse

Tous les systèmes de médiation que nous avons présenté manipulent des données semi-structurées et hétérogènes. Le tableau 2.1 synthétise les propriétés de ces systèmes. Les principales différences entre celles-ci se situent au niveau du modèle de représentation des données, certains transforment les données XML en tuples relationnels pour la représentation interne, d'autres utilisent le modèle objet pour représenté les données semi-structurées, enfin, certains sont des systèmes «*tout-XML*» (langage XQuery et modèle de données XML). Parmi tous ces systèmes, seul *LiquidData* propose une architecture tout-XML.

Nom	**Langage**	**Modèle de données**	**Approche**
TSIMMIS	OEM-QL (proche OQL)	OEM	GAV
GARLIC	GQL (proche que SQL)	Objet	GAV
DISCO	OQL	Objet (ODMG)	GAV
XPeranto	XGQM	Relationnel	GAV
AGORA/LeSelect	XQuery	Relationnel	LAV
LiquidData	XQuery	XML	GAV
Piazza	SQL & XQuery	Relationnel & XML	GAV-LAV
SOMEWHERE	OWL PL	OWL	GAV

Tab. 2.1 – Propriétés des systèmes de médiation semi-structurées

2.2 Traitements de requêtes XQuery

La plupart des Systèmes de Gestion de Base de Données (SGBD) s'appuient sur des structures régulières, relationnelles ou objets, qu'ils peuvent manipuler. Les caractéristiques de ces systèmes se basent sur des schémas fixes auxquels les données doivent se conformer. Cela permet de simplifier le stockage et l'interrogation des données, de plus, les méthodes d'interrogation sont plus facilement efficaces. Toutefois, les types de données flexibles demandées par l'utilisateur ne sont parfois pas représentables dans ces modèles. Le modèle n'est pas fixe (structure, schéma, données, typage, multivaluation). Les modèles relationnels et objets ne conviennent pas dans ce contexte. C'est pourquoi le modèle semi-structuré a été introduit pour répondre aux demandes liées à Internet, non contraint à des structures rigides. Les caractéristiques des données semi-structurées sont décrites dans [Buneman 1997][Abiteboul *et al.* 1997].

Pour représenter les données semi-structurées, un large consensus s'est fait autour de XML. Le langage XML ou **eXtended Markup Language** [Bray *et al.* 1997] est un standard proposé par le W3C [W3C 1994]. Il normalise un langage d'échange, ouvert, solide et capable d'assurer l'interopérabilité avec les technologies *Internet*, ainsi qu'avec les systèmes d'information d'entreprises. En effet, trois tendances nouvelles tirent aujourd'hui le développement des systèmes d'informations vers le futur :

- *l'ouverture au Web et le développement de nouvelles architectures* ;
- *le développement des échanges* : échanges intéropérables, A2A[1], B2B[2] ;
- *l'intelligence accrue des systèmes* : entrepôts de données (*Datawarehouse*), fouille de données (*datamining*).

Le langage XML permet de répondre à ces exigences. De plus, pour couvrir un plus large ensemble de problèmes, les technologies XML sont décomposées en domaines spécialisés : DTD, XML-schema, Namespace, XPath, XQuery.

```
<catalog>
  <book genres="roman" isbn="123456789">
    <title> L'assasin du roi</title>
    <author>Robin Hobb </author>
    <price currency="euros"> 16,26 </price>
  </book>
  <book genres="roman">
    <title> La nef du crépuscule </title>
    <author>Robin Hobb </author>
  </book>
  <book genres="roman">
    <title> Les secrets de Castelcerf  </title>
  </book>
</catalog>
```

Fig. 2.3 – Exemple de document XML

La figure 2.3 nous montre un document XML qui nous sert d'exemple dans cette thèse. Il contient les informations décrivant trois romans. Les éléments sont représentés en gras, les attributs en italique et le texte normal représente les données. Ainsi, ces trois livres : «*L'assassin du roi*», «*La nef du crépuscule*» et «*les secrets de castelcerf*» sont représentés par trois éléments «*book*», dans lesquelles nous pouvons identifier le titre, l'auteur et le prix du livre.

2.2.1 Aperçu de XQuery

Un document XML représente des données semi-structurées. L'interrogation de ce document doit nous permettre de récupérer certaines informations, mais aussi de produire un résultat. A cette fin, un langage d'interrogation est nécessaire pour

[1] Application to Application
[2] Business to Business

récupérer les informations contenues dans un document XML.

De nombreux langages de requêtes ont fait objet de travaux pour répondre aux prérogatives des données semi-structurées. Parmi ceux-ci nous pouvons distinguer une extension du langage objet *OQL* [Christophides *et al.* 1994] pour les traversées de graphes généralisés et les recherches de contenus, *SGMLQL* [Maitre *et al.* 1997] réalisant des requêtes sur des documents SGML, LOREL / OEM-QL [Abiteboul *et al.* 1997] permettant la construction de graphes généralisés sur le modèle de données OEM. Puis, *XPath* [W3C 1999], un langage de requête simple pour exprimer des chemins dans un document a été proposé, et est devenu un standard du W3C. Sur ce standard ont été proposés plusieurs langage de requêtes plus complexe tel que *XML-QL* [Deutsch *et al.* 1998] (AT&T), *XQL* [Robie *et al.* 1998] (Microsoft), *QUILT* [Chamberlin *et al.* 2000], *CDuce* [Benzaken *et al.* 2003][Benzaken *et al.* 2004][Benzaken *et al.* 2005] un langage puissant pour l'expression du typage de données, *XyQL* [Aguilera *et al.* 2000] (Xyleme/INRIA) pour l'interrogation de documents textuels, et enfin *XQuery* [W3C 2006a] qui est devenu le standard du *W3C* pour l'interrogation de documents XML.

XQuery est le langage aujourd'hui utilisé pour l'interrogation de documents XML. Il est issu de Quilt [Chamberlin *et al.* 2000] et reprend les avantages de XQL [Huck et Macherius 2000], XML-QL [Deutsch *et al.* 1998] et XPath [W3C 1999] (détaillé en annexe B). Le premier document public de travail du W3C date du 15 février 2001, depuis, plusieurs versions ont été proposées. La version étudiée ici est celle de juin 2006, la grammaire de XQuery est détaillée en annexe C.

Le langage XQuery permet d'exprimer des sélections de chemins dans un document XML grâce à XPath. Chaque sélection est assignée à une variable définissant un ensemble d'arbres. Ces ensembles peuvent être manipulés par des contraintes, des ordonnancements, des constructions de résultat, des imbrications de requêtes, des quantificateurs, des agrégats, des séquences, des opérations ensemblistes et conditionnelles. Les spécifications du langage sont disponibles sur le site du [W3C 2006a] et détaillées en annexe C. De plus, un extrait des cas d'usage du [W3C 2006b] sont disponibles en annexe F.

Afin d'illustrer le langage, le tableau 2.2 donne trois exemples de requêtes XQuery permettant d'interroger le document XML présenté dans la figure 2.3. Chacune d'elle permet de donner le livre dont l'auteur est «*Robin Hobb*». Les deux premières requêtes vérifient si le titre du livre est «*L'assassin du roi*». La troisième requête introduit la notion d'imbrication des requêtes, celle-ci demande les livres dont le titre est «*L'assassin du roi*» et le prix supérieur à 14 euros, nous pouvons constater que le résultat donne un élément «*catalog*» supplémentaire *vide* correspondant à un élément du document source dont l'auteur est «*Robin Hobb*».

Pour ces trois requêtes, il n'est évident d'identifier les caractéristiques principales de celles-ci au premier coup d'oeil. Afin de permettre une meilleure visualisation et une catégorisation des requêtes, un modèle de représentation est nécessaire. Ainsi, nous pourrons identifier simplement une requête à l'aide son modèle. Nous étudions

Requête XQuery	Résultat en XML
for $i in doc(”cat.xml”)/catalog where $i//author = ”Robin Hobb” and $i/book/title = ”L’assassin du roi” return <catalog>{$i/book}</catalog>	<catalog> <book genres=”roman” isbn=”123456789”> <title>L’assassin du roi</title> <author>Robin Hobb</author> <price currency=”CDN”>16,26</price> </book> </catalog>
for $i in doc(”cat.xml”)/catalog[.//author = ”Robin Hobb”] where $i/book/title = ”L’assassin du roi” return <catalog>{$i/book}</catalog>	<catalog> <book genres=”roman” isbn=”123456789”> <title>L’assassin du roi</title> <author>Robin Hobb</author> <price currency=”CDN”>16,26</price> </book> </catalog>

for $i in doc(”cat.xml”)/catalog[.//author
= ”Robin Hobb”]
return
<catalog>
{
  for $j in doc(”cat.xml”)/catalog/book
  [.//title = ”L’assassin du roi”]
  where $j/price > 14
  and $i/book/@id = $j/@id
  return $j
}
</catalog>
	<catalog>
	  <book genres=”roman” isbn=”123456789”>
	  <title>L’assassin du roi</title>
	  <author>Robin Hobb</author>
	  <price currency=”CDN”>16,26</price>
	  </book>
	</catalog>
	<catalog>
	</catalog>

Tab. 2.2 – Exemples de requêtes XQuery

les différents modèles de représentation dans la section 2.2.2, les requêtes présentées dans le tableau 2.2 y sont représentés.

De plus, nous pouvons constater que le résultat des deux premières requêtes sont identiques. En effet, les spécifications du W3C concernant les requêtes XQuery peuvent introduire des similitudes dans l’évaluation des requêtes. Il est alos possible d’exprimer une requête de différentes manières. Pour donner une forme unique aux requêtes, et ainsi de faciliter la représentation des requêtes, des règles de canonisation sont nécessaires. Elles donnent les transformations équivalentes permettant de donner une forme canonique aux requêtes. Nous donnons des règles de canonisation dans la section 2.2.3.

2.2.2 Représentation de requêtes

Dans les systèmes de gestion de base de données, une structure interne au système est nécessaire pour représenter les requêtes XQuery. Chaque système apporte sa structure particulière et les besoins du système donne à la représentation ses spécificités. Dans le cadre du langage *XQuery*, de nombreuses solutions ont été proposées, toutefois, celle qui a émergé se nomme le *Tree Pattern Matching* : filtrage par motif d'arbres. Nous allons étudier l'évolution de cette structure représentant les requêtes *XQuery*.

Tree Patterns Le *tree pattern matching* proposé dans l'algèbre TAX [Jagadish *et al.* 2001] a donné un ensemble de propositions autour des **Tree Pattern Queries** (ou *TPQ*) [Amer-Yahia *et al.* 2001][Ramanan 2002][Shasha *et al.* 2002][Lakshmanan *et al.* 2004][Yao et II 2004][Manolescu *et al.* 2004][Moro *et al.* 2005]. Le principe réside dans la modélisation des relations d'ascendance (parent/enfant, ancêtre/descendant) entre les différents éléments requis par une requête. Une formule regroupe l'ensemble des motifs qui doivent être appliquée à un document. Si celui-ci respecte l'ensemble de tous les motifs (nom d'élément, liens d'ascendance, prédicats), il est alors filtré par le *TPQ* et utilisé comme résultat.

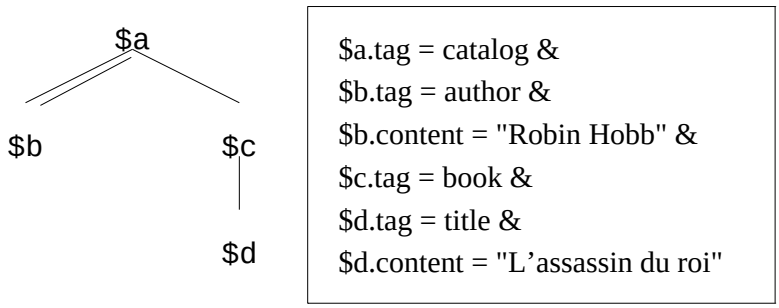


FIG. 2.4 – Exemple de Tree Pattern Query (TPQ)

La figure 2.4 illustre le *Tree Pattern Query* correspondant aux deux premières requêtes du tableau 2.2. Nous pouvons y observer les liens d'ascendance de trois nœuds, auxquels est associée une formule donnant le motif correspondant à chaque nœud. La formule définit le motif des noms de balises que le document doit contenir, tel que «catalogs» pour $a, «title» pour $b, «book» pour $c et «author» pour $d. Les motifs $c et $d doivent contenir un texte précis pour être filtrés. Les liens simples représentent une relation parent/enfant, tandis que le lien double une relation ancêtre/descendant entre les deux éléments.

Ce modèle de représentation de requêtes ne permet pas de modéliser plusieurs motifs d'arbre à la fois. Il n'est donc pas possible de modéliser une jointure, ni une imbrication de requêtes. La troisième requête du tableau 2.2 illustre bien cette incapacité. Il est donc nécessaire d'étendre le Tree Pattern matching pour intégrer ces possibilités.

Generalized Tree Pattern Par la suite, le principe des *Tree Patterns* (motifs d'arbre) a été amélioré en y ajoutant quelques propriétés supplémentaires. En effet, celles-ci ne permettent pas de représenter la complexité du langage *XQuery* comme les jointures, les imbrications, l'optionalité ou les quantificateurs. Ainsi, les **Generalized Tree Pattern** (ou *GTP*) [Chen 2004] ont été créés pour améliorer le *Tree Pattern Matching*. Les liens d'optionalité sont représentés en pointillé. Des motifs d'arbre sont associés à l'aide d'une formule pour représenter les jointures. Les imbrications sont représentées à l'aide d'annotations.

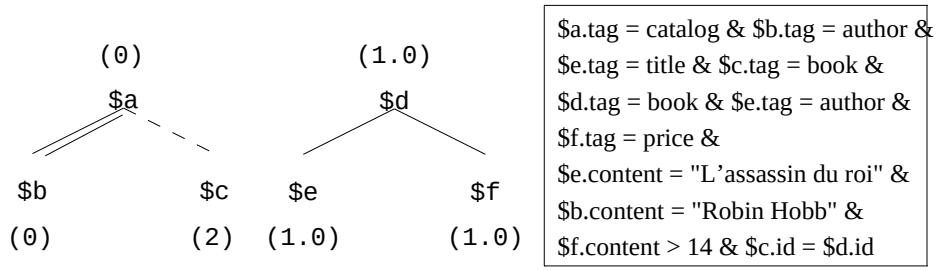


FIG. 2.5 – Exemple de Generalized Tree Pattern (GTP)

La figure 2.5 illustre le *GTP* correspondant à la troisième requête du tableau 2.2. Il contient deux *Tree Patterns* donnant une liste de livres dont l'auteur est «*Robin Hobb*», le second motif d'arbre demande un prix est supérieur à 14 euros et un titre de livre correspondant à «*L'assassin du roi*». Les deux motifs d'arbre sont reliés par des niveaux d'imbrication, annotés sur les nœuds, et par la jointure définie dans la formule. La formule globale contient les motifs associés aux noms des éléments, noms des attributs, prédicats sur valeurs et prédicats de jointures.

XML Access Modules Sur ces propositions s'est ajoutée une nouvelle algèbre basée sur les motifs d'arbre : les **XML Access Modules** (ou XAMs) [Arion *et al.* 2005a][Arion *et al.* 2005b][Arion *et al.* 2006]. Cette algèbre, spécifique à la description d'information contenues dans des structures de stockage XML (en particulier les vues indexées ou matérialisées), permet de modéliser une requête sur des données semi-structurées. Le formalisme des opérateurs de cette algèbre permet de retrouver efficacement les données stockées.

La figure 2.6 illustre un XAM donnant les liens structurels entre chaque élément (jointures), leurs noms de balise, ainsi que la projection de leurs valeurs. Ainsi, cet exemple filtre une collection de livres ; les informations retournées sont l'année, le nom de l'auteur et le titre du livre.

Synthèse Les techniques basées sur les motifs d'arbre (GTP) permettent de représenter une requête *XQuery*, en particulier les associations provenant des chemins *XPath* contenus dans la requête. Ils permettent de modéliser des imbrications, des

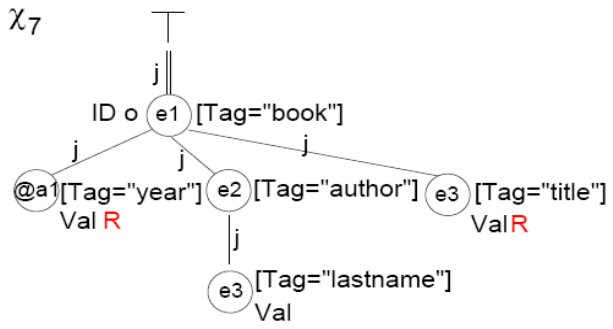


FIG. 2.6 – Exemple de XML Access Module (XAM)

jointures, des projections d'éléments, l'optionalité des éléments. Toutefois, par rapport à la grammaire du langage *XQuery*, certaines caractéristiques n'apparaissent pas dans ces modèles. En effet, les requêtes *XQuery* peuvent exprimer :
– des collections sources ;
– des reconstructions de résultat contenant des balises ;
– des définitions de fonctions et leurs utilisations ;
– des définitions de clause *let* ;
– des opérations ensemblistes et séquentielles ;
De plus, les GTP proposent une représentation des motifs par une formule difficile à lire, ce qui ne facilite pas une représentation intuitive de requêtes. Élaborer une représentation de requêtes capable de couvrir la grammaire *XQuery* reste donc une tâche complexe, certaines caractéristiques du langage n'étant pas modélisées, voire non reconnues.

2.2.3 Canonisation de requêtes

Grâce au fort potentiel du langage XQuery, il est possible d'écrire de nombreuses requêtes différentes. Mais cette trop grande liberté grammaticale nuit à l'interprétation du langage dans les systèmes de gestion de données XML. En effet, pour démontrer qu'une requête XQuery est valide, il faut étudier l'intégralité du langage. Or XQuery et XPath possèdent une grammaire complexe, où deux expressions peuvent être identiques. Afin de simplifier l'ensemble de définition de XQuery et la représentation de certaines expressions, [Deutsch *et al.* 2004] et [Chen 2004] proposent des règles de réécriture de requêtes permettant de restructurer les requêtes tout en préservant les résultats.

NEXT Les règles de transformation proposée par [Deutsch *et al.* 2004], basées sur les travaux de minimisation de requêtes de [Amer-Yahia *et al.* 2001] et [Ramanan 2002], dans NEXT, s'inspirent des '*group-by*' utilisés dans le langage OQL, nommée *OptXQuery*. Afin d'éliminer les redondances dans le parcours des éléments dans un document, NEXT restructure les requêtes pour traiter plus efficacement les requêtes

fortement imbriquées.

N°	Requête initiale	Requête minimisée
R2	for $V in (for V_1$ in E$_1$ return E$_2$) return E$_3$	→ for V_1$ in E$_1$ return for $V in E$_2$ return E$_3$
R10	some $V in distinct-values (E) satisfies C	→ some $V in E satisfies C
R11	$V in (/|//)C	→ for V_1$ in $V(/|//)C return V_1$

Tab. 2.3 – Exemples de règles de minimisation dans NEXT

Le tableau 2.3 illustre trois règles de minimisation proposées pour les requêtes internes à NEXT, les *OptXQuery*. Celles-ci modifient la requête pour la préparer vers l'algèbre ; les *Next queries*, résolvant l'imbrication de requêtes, sont le point fort de ce système. Le problème lié à ces restructurations provient du fait que les requêtes obtenues ne sont plus des requêtes XQuery (mais des OptXQuery) contenant des clauses *group-by*, ne respectant plus le standard (il s'agit d'une structure interne à NEXT). Dans le cadre de la médiation, le dialogue en requête XQuery devient alors difficile. De plus, la représentation toutes les requêtes XQuery n'est pas non plus possible dans ce cas.

GTP Les GTP [Chen 2004] que nous avons présenté précédemment sont associés à un ensemble de règles de transformation de requêtes *XQuery*. Celles-ci sont utilisées dans un but de structuration de la requête, afin d'obtenir une requête *XQuery* canonique. Cette requête canonique peut alors être analysée pour obtenir un modèle simple et unique correspondant à la requête canonisée. La grammaire d'une requête canonique présentée dans le tableau 2.4 est assez restreinte mais permet de reconnaître un sous-ensemble de *XQuery* assez conséquent.

expr : := (for $$fv_1$ in $range_1$, ... , $$fv_m$ in $range_m$) ?
(let $$lv_1$:= "(" $expr_1$ ")", ... , $$lv_n$:= "(" $expr_n$ ")") ?
(where φ) ?
return
<result>
$< tag_1 >\{arg_1\}< /tag_1 > ... < tag_n >\{arg_n\}< /tag_n >$
< /result>

Tab. 2.4 – XQuery Canonique dans les GTP

Ainsi, nous obtenons une syntaxe XQuery bien particulière, qui nous permet de caractériser simplement une famille de requêtes XQuery.

Dans la thèse de [Chen 2004], il est démontré qu'une requête XQuery, dont la syntaxe respecte une partie de celle présentée en annexe C, peut toujours être traduite en une requête XQuery canonique. Les lemmes ci-dessous permettent de transformer celles-ci cas par cas. On dit que deux requêtes sont **équivalentes** si pour un même ensemble de départ l'ensemble des résultats des deux requêtes sont identiques et ordonnées.

1. Une expression XPath peut contenir des restrictions sous forme de filtres (voir annexe B). Ceux-ci peuvent être remplacés par des variables intermédiaires auxquels sont associés le(s) prédicat(s) du filtre dans la clause *where*. L'exemple du tableau 2.5 illustre la transformation du filtre en un nouveau prédicat dans la clause *where*.

Requête XQuery	**Requête XQuery Canonique**
for $i in doc("cat.xml")/catalog/book [@isbn="12351234"]/title return {$i}	for $j in doc("cat.xml")/catalog/book **for $i in $j/title** where **$j/@isbn = "12351234"** return {$i}

Tab. 2.5 – Requête avec filtre

2. Une expression FLWR contenant des requêtes imbriquées peut être réécrite en une expression équivalente tel que les expressions FLWR soient déclarées dans des clauses *let*. Afin d'éviter les problèmes d'imbrication de requêtes à répétition, il faut définir une nouvelle clause *let* contenant l'expression imbriquée, et redéfinir celle-ci par le nouveau nom de variable. L'exemple donné dans le tableau 2.6 redéfinit une requête imbriquée dans la clause *let* : «*let $l := (...)* », et la valeur de retour devient *$t*.

Requête XQuery	**Requête XQuery Canonique**
for $i in doc("cat.xml")/catalog/book return <livre> {for $j in $i/title return {$j}} </livre>	for $i in doc("cat.xml")/catalog/book **let $l :=** (for $j in $i/title return {$j}) return <livre>**{$l}**</livre>

Tab. 2.6 – Transformation de requête imbriquée

3. Une expression FLWR contenant un quantificateur «every» peut être transformée en une expression équivalente utilisant une expression de quantité. La syntaxe XQuery définit les quantificateurs «every» comme un prédicat de la formule booléenne φ. Elle permet de vérifier si toutes les instances d'une variable vérifient le prédicat donné. La requête initiale du tableau 2.7 retourne tous les livres dont tous les prix[3] de ventes sont strictement supérieurs à 15 euros. Afin de simplifier et de canoniser cette requête, il faut créer une clause

[3] on supposera qu'un livre peut avoir plusieurs prix

let contenant les livres dont les prix sont inférieurs ou égaux à 15 euros. Si le nombre d'arbres de cette clause est supérieur à 0, alors l'instance d'arbre de $i ne satisfait pas le prédicat "every" et l'instance n'est pas retournée.

4. De la même manière, une expression FLWR contenant un quantificateur d'existence «some» peut être transformée. La transformation est la même, mais il faut compter s'il y a au moins une instance d'arbre qui vérifie la condition.

5. Les agrégats définis dans une expression FLWR peuvent être réécrits dans une clause *let*, associée à une nouvelle variable. Cette variable remplace la fonction d'agrégat là où elle était initialement.

Requête XQuery	**Requête XQuery Canonique**
for $i in doc("cat.xml")/catalog/book where every $s in $i/price satisfies $s > 15 return {$i}	for $i in doc("cat.xml")/catalog/book **let $l :=** (for $j in $i/price where **$j <= 15** return {$j}) where **count($l) = 0** return {$i}

Tab. 2.7 – Transformation du quantificateur "every"

[Chen 2004] démontre au final qu'une expression FLWR quelconque peut toujours être transformée en une requête XQuery canonique. Cette canonisation repose sur plusieurs règles de transformations permettant de transformer une requête *XQuery* en une requête *XQuery* canonique.

La requête XQuery donnée en exemple dans le tableau 2.8 est transformée en une requête *XQuery* canonique dans le tableau 2.9. Les lemmes de désimbrication, déplacement d'agrégat et de filtres sont appliqués.

```
for $y in collection ("reviews")/review[. contains ("daulphin")]/book
where $y/price > 15
return
< result >
        {$y/@isbn}
        {$y/price}
        < nb_titles >{
                for $z in collection ("books")/book
                where $z/@isbn = $y/@isbn
                return
                        {count ($z/title)}
        }< /nb_titles >
< /result >
```

Tab. 2.8 – XQuery non canonisée

```
for $x in collection ("reviews")/review,
        $y in $x/book
let $l1 :=
(
        for $z in collection ("books")/book
        let $l2 := count ($z/title)
        where $z/@isbn = $y/@isbn
        return
                {$l2}
) where
        $x contains ("daulphin")
        and $y/price > 15
return
< result >
        {$x/@isbn}
        {$y/price}
        < nb_titles >{$l1}< /nb_titles >
< /result >
```

Tab. 2.9 – XQuery canonisée

Nous avons pu voir dans cette partie, grâce aux règles de minimisation de [Deutsch *et al.* 2004] et de canonisation de [Chen 2004], que les requêtes *XQuery* pouvaient être manipulées tout en préservant leur sens. Toutefois, elles ne correspondent pas à tous les pré-requis du contexte de médiation et d'optimisation sur des requêtes *XQuery* globale. La grammaire simplifiée du langage *XQuery* de [Chen 2004] se rapproche de nos besoins. Toutefois ces requêtes canoniques ne peuvent supporter toute la richesse du langage XQuery, supprimant ainsi certaines propriétés intéressantes de ce langage comme les :
- opérations ensemblistes ;
- opérations conditionnelles ;
- séquences ;
- définitions de fonctions.

Il nous faudra donc étendre la définition de ces règles de canonisation pour donner une grammaire canonisée plus générale du langage *XQuery*, permettant de couvrir un ensemble plus conséquent des requêtes *XQuery* existantes.

2.3 Optimisation Extensible

Les systèmes d'optimisation extensible sont des systèmes de gestion de données permettant d'ajouter des fonctionnalités dans l'optimiseur de requêtes et plus généralement au SGBD. En effet, ils permettent une flexibilité dans la façon de concevoir un système de gestion de base de données. Le terme extensible réfère à l'idée qu'un système peut être étendu avec de nouvelles capacités de traitement pour répondre aux besoins de différents systèmes [Carey et Haas 1990].

Dans cette thèse, nous souhaitons faciliter l'extension de l'optimiseur de requêtes dans un système de médiation. L'extension de l'optimiseur doit favoriser l'`amélioration` des performances et la `personnalisation` du médiateur. Cette personnalisation permet d'adapter le système en fonction de ses besoins (sources spécifiques,

données particulières, intégration de modules différents...). Ainsi, l'intégration d'un optimiseur extensible facilitera l'`adaptabilité` du système de médiation.

Une approche pour étendre un système de gestion de bases de données est de fournir une collection d'outils qui seront utilisés pour bâtir un nouveau système de base de données. Une approche plus ouverte est de fournir un système de gestion de base de données qui peut être facilement modifié pour lui incorporer de nouvelles capacités. Dans cette section nous allons revoir les principes des SGBD extensibles, quelques systèmes extensibles, mais en particulier leur optimiseurs.

2.3.1 Base de l'extensibilité

Le premier exemple d'architecture extensible est le système *Eris* proposé par *Reiss* [Reiss 1982a] [Reiss 1982b]. Le but d'*Eris* est de supporter de nouveaux opérateurs et de nouvelles architectures, structures de données et techniques d'évaluation au niveau implémentation. La flexibilité d'*Eris* est proposée à travers une architecture de flux de données appelée «*modèle de chemins*». La modélisation dans ce système est basée sur un graphe de flux de données dont les nœuds représentent les opérateurs et les arcs sont les chemins de données. Les extrémités de ce graphe accèdent à la base de données et les fonctions de coût sont associées à chaque opération. L'optimisation de requêtes est traitée en trois étapes :

1. Normalisation de l'expression des sources ;
2. Traduction de l'expression des sources en graphe de chemins, à l'aide d'une programmation dynamique ;
3. Transformation des graphes de chemins à l'aide d'optimisations locales.

L'extensibilité de ce système provient d'un modèle efficace, les «modèles de chemins», ainsi que la généralisation des étapes d'optimisation.

Il existe deux types d'extensibilités dans les optimiseurs de requêtes : les extensibilités d'espace de recherche, et les extensibilités de stratégies de recherche ([Lanzelotte et Valduriez 1991][McKenna *et al.* 1996][Kabra et DeWitt 1999]). Beaucoup de systèmes utilisent des règles de réécritures pour transformer les requêtes, permettant ainsi une optimisation sur les expressions des requêtes [Graefe 1987] [Sieg 1989] [Finance et Gardarin 1991] [Piraresh *et al.* 1992].

2.3.2 Systèmes extensibles

Nous allons ici étudier quelques systèmes avec optimisation extensible. Une synthèse sera proposée à la fin de cette section pour comparer ces architectures.

Exodus. Le système de gestion de base de données *EXODUS* [Carey *et al.* 1990] est un ensemble d'outils permettant de spécifier un modèle de données et la création

d'une base de données orientée objet. Ce système contient un générateur d'optimiseur [Graefe 1987] [Graefe et DeWitt 1987] qui construit un optimiseur pour un modèle spécifié lors de la génération. Ce générateur utilise les informations sur le modèle de données, les opérations, les méthodes d'accès, le modèle de coût et les transformations applicables. Les règles de l'optimiseur sont des règles de réécritures de requêtes décrivant des transformations d'opérations algébriques, ou les règles d'implémentation décrivant les méthodes d'accès pour une opération. L'optimiseur généré applique les règles données à un arbre d'évaluation de requêtes pour trouver la séquence d'opérations la moins coûteuse lors de l'exécution. L'optimiseur utilise une stratégie de recherche «*Best-First*» basée sur le meilleur bénéfice lors du choix d'une règle. Les règles qui peuvent générer un coût plus grand sont tout de même considérées à partir du moment où elles sont capables de permettre d'accéder à des règles qui n'auraient pas pu être appliquées précédemment et ainsi produire un meilleur plan (méthode du recuit-simulé). L'extensibilité du système est produite par la regénération d'optimiseurs avec des informations différentes.

Volcano. Le système *Volcano* est un générateur d'optimiseur utilisant les principes d'*EXODUS* tout en ajoutant des heuristiques et de la sémantique pour guider la recherche de transformations. La caractéristique de cette architecture est de proposer une forte extensibilité *via* la possibilité d'intégrer les heuristiques, un support d'extensibilité pour les types de données, un support pour les modèles de coût modulable [Graefe et McKenna 1991] [Graefe et McKenna 1993] et sa stratégie de recherche basée sur la programmation dynamique dirigée (*Directed Dynamic programming*). Comme pour *EXODUS*, le processeur de requête est basé sur des règles de transformations algébriques. Toutefois, la représentation interne de requêtes et les stratégies de recherches utilisées sont plus efficaces dans *Volcano*. L'idée principale est d'explorer uniquement les sous-requêtes ou plans qui participent effectivement à une expression plus large. *Volcano* est capable de supporter des fonctionnalités de parallélisme dans les opérateurs, par contre, l'optimisation du parallélisme est séparée de l'architecture d'optimisation [Graefe *et al.* 1991][Graefe 1994].

```
1: declaration part:
2:    %operator 2 join
3:    %method 2 hash_join loops-join cartesian_product
...
4: rule part:
5:    join(1,2) ->! join(2,1),
6:    join(1,2) by hash_join(1,2),
```

Tab. 2.10 – Exemples de règles définis pour Volcano

L'exemple de la figure 2.10 illustre la définition de transformations dans Volcano. Les règles de transformation et d'implémentation (*Rule Part*) définissent les permutations des jointures. L'opérateur join (ligne 2) et les trois méthodes *hash_join,*

loops_join, cartesian_product (ligne 3) ont deux entrées pour chacun ; la ligne 5 représente la commutativité de jointures (règle de transformation) et la ligne 6 représente que *hash_join* est un algorithme implémentant l'opérateur *join* (règle d'implémentation). Ainsi, le programmeur doit définir les règles de transformation, mais aussi le code des fonctions de coût et d'autres traitements, qui apparaît sous forme de procédures en langage C.

Starburst. Le système de gestion de bases de données relationnelles *Straburst* [Widom 1996] est basé sur un modèle extensible. Son point fort est son habilité à introduire de nouveaux types de données et opérations (stockage, méthodes d'accès, méthodes de traitement interne) [Haas *et al.* 1989][Schwarz *et al.* 1986]. Le système travaille sur les requêtes grâce à une représentation de celles-ci sous forme de graphes, puis utilise des règles de transformation, puis planifie une stratégie d'évaluation pour la requête [Haas *et al.* 1989][Piraresh *et al.* 1992].

Les règles de réécriture de requêtes sur le graphe sont appliquées sous conditions [Piraresh *et al.* 1992]. Ces règles sont écrites dans un langage procédural et peuvent être groupées en ensemble de règles. Le processeur de règles peut traiter en ensemble de règles selon un ordre séquentiel ou prioritaire. Durant l'étape d'exécution de plan, les opérateurs d'accès à la base de données sont des opérateurs implémentés en langage de bas niveau permettant d'intégrer des règles grammaticales [Lohman 1988]. La construction de ces opérateurs propose une forte extensibilité dans les stratégies de recherches lors d'accès à une base de données.

create rule name **on** table
when triggering-operations
[**if** condition]
then action-list
[**precedes** rule-list]
[**follows** rule-list]

Tab. 2.11 – Syntaxe de règle de Starburst

La syntaxe du langage de règle présenté dans le tableau 2.11 s'exprime de la même manière que les triggers du langage SQL. Cela permet au processeur de règles de s'intégrer simplement dans l'architecture d'optimisation au niveau de la syntaxe d'expression.

ESPRIT EDS. Le Projet ESPRIT EDS [Finance et Gardarin 1991] [Finance et Gardarin 1992] est un optimiseur extensible a base de règles de transformation définie à l'aide d'un langage de définition de règles très complet. Ce langage puissant, permet d'intégrer les conditions d'application, des fonctions de coût, des méthodes d'exécution et de réécriture. Le principe de la stratégie de recherche est d'organiser les règles de transformation en modules créant ainsi des méta-règles de séquence

<rule name> :
IF <Term>
[**under** <Constraint>]
[**delta_cost** = <Function Expression>]
THEN [**execute** <Method List>]
[**rewrite** <Term>]

TAB. 2.12 – Syntaxe de règle du projet ESPRIT EDS

de règles de transformation, et ainsi de réduire l'espace de recherche. Le but de l'optimiseur est de choisir judicieusement les méta-règles d'optimisation.

Le tableau 2.12 illustre le langage de définition de règles du projet ESPRIT EDS, une règle se lit de la manière suivante : «*Si le terme de la clause* `IF` *apparaît dans le plan de requête sélectionnée sous l'ensemble de contraintes donné, alors il est récrit selon le terme de la clause* `THEN` *associée à la liste de méthodes. L'expression* `delta_cost` *donne la variation de temps estimée induit par l'application de la règle de réécriture ; elle est traitée avant la réécriture.*»

EPOQ. *EPOQ* [Bjørnestad 1994][Mitchell *et al.* 1993] est un optimiseur extensible orienté objet. Parallèle aux travaux d'*Exodus*, le principe de l'extensibilité de cet optimiseur repose sur la planification de règles groupées par régions d'optimisation. Par l'intermédiaire de ces régions, une requête et un but forment une tâche à accomplir. L'optimiseur choisit la séquence de régions qui lui convient le mieux. Pour atteindre un but, la région d'optimisation parcourt l'ensemble des sous-régions qui la compose en combinant chaque exécution. Une sous-région prévient sa région si elle est capable ou pas de s'appliquer (et ainsi de suite récursivement sur les sous-régions). Un «*log*» permet de garder une trace des régions appliquées permettant un retour en arrière. Ainsi, *EPOQ* est un système de planification de règles.

De plus, le système permet à une région d'avoir un contrôle sur ses sous-régions par l'intermédiaire de messages d'arrêts. La condition d'arrêt de la région n'est pas alors que dépendante de ses sous-régions, mais aussi de son propre choix. Le choix final de l'optimiseur est la construction d'un résultat pour la requête. Cette décision est liée au choix des requêtes alternatives générées.

Les requêtes sont traduites dans la représentation de requêtes internes *qTree* (Query Tree Interface) [Lo 1992], dans laquelle sont intégrées des annotations de coût. Cette représentation est une forme intermédiaire avant d'être représentée en une *Query-Rep* qui est la représentation utilisée par l'architecture EPOQ. Mais toute requête représentée est une qTree.

La différence majeure entre cette approche et les autres présentés est que le système d'optimisation orienté objet *EPOQ* permet une optimisation de son propre processus. Tandis que les autres architectures proposent des stratégies fixes, ou à implémenter par l'intermédiaire de règles de transformation.

Extension de Sybase. Sybase est un système de gestion de base de données relationnel. [Andrei et Valduriez 2001] propose une extension de l'optimiseur de Sybase. Le principe de cette extension est que la plupart des optimiseurs obtiennent un trop large espace de recherche et peuvent alors faire une erreur décisionnelle dans leur stratégie. Ainsi, [Andrei et Valduriez 2001] propose un langage de requêtes sur des plans abstraits : **AP**, pour palier les problèmes de décision. Ce langage basé sur une algèbre relationnelle physique, est utilisé à la fois par l'optimiseur pour décrire le plan sélectionné et par l'utilisateur pour orienter les choix de l'optimiseur.

Les opérateurs du langage *AP* décrivent les transformations équivalentes des algorithmes relationnels supportés par l'optimiseur, et les choix possibles. Comme une expression relationnelle peut être représentée sous forme d'un arbre algébrique, AP en est la représentation textuelle. Ainsi, nous pouvons exprimer chaque requête SQL sous forme d'APs contenant les algorithmes disponibles pour chaque opérateur. Le principe d'utilisation de l'AP est de s'ajouter à un optimiseur classique. L'optimiseur génère des requêtes, tandis que grâce à l'AP d'autres possibilités de requêtes pouvant être rejetés seront générées, permet à l'optimiseur de retourner dans un choix de requêtes plus optimal.

Cette approche se différencie des autres optimiseurs par sa capacité à s'intégrer à un optimiseur existant. Elle permet d'élargir l'espace de recherche généré par cet optimiseur, permettant d'obtenir des arbres algébriques optimaux, et d'éviter de graves erreurs de choix d'optimisation.

OPT++. Basé sur les travaux de l'extensibilité de stratégie de recherche ([Lanzelotte et Valduriez 1991][McKenna *et al.* 1996]), les auteurs d'OPT++ [Kabra et DeWitt 1999] proposent de séparer la représentation de plan logique et de plan physique. Les opérations de manipulation de plan de requête sont représentées sous forme de classes, et non par des méthodes comme fait précédemment ([Lanzelotte et Valduriez 1991]). Ainsi, les transformations possibles se font sur le plan logique, le plan physique et des insertions d'*enforcers* dans le plan. Étant donné que les classes sont abstraites, la spécialisation de l'implémentation d'un opérateur permet l'optimisation du plan par l'intermédiaire de différentes stratégies de recherche.

Le système OPT++ se différencie des autres approches en permettant d'étendre la stratégie de recherche par l'intermédiaire de l'optimisation du plan logique et du plan physique.

2.3.3 Synthèse

Nous avons passé en revue plusieurs techniques d'optimisation extensible ; celles-ci se basent sur des définitions de règles de transformation, d'heuristiques, d'ensembles de transformations, de stratégie de recherche et des modèles de coût. Ces systèmes relationnels ou objets permettent de modifier l'espace de recherche généré par l'optimiseur et ainsi de trouver une solution optimale. Cette solution optimale est définie

par la stratégie de recherche qui génère un espace de recherche optimal. Le tableau 2.13 synthétise les modèles de données utilisées par chaque architecture, ainsi que leur(s) stratégie(s) de recherche et sa base d'extensibilité.

Nom	**Modèle de données**	**Stratégie de recherche**	**Extensibilité**
EXODUS [Carey *et al.* 1990]	Objet	Best-First + recuit simulé	Transformation d'opérateurs algébriques, implémentation d'opérateurs
Volcano [Graefe et McKenna 1991]	Relationnel, Objet	Heuristiques	Transformation et implémentation d'opérateurs, définition d'heuristiques et de modèles de coût, programmation dynamique dirigée
Starburst [Widom 1996]	Relationnel	Bottom-up	Types de données, Opérations, Langage de Règles
ESPRIT EDS [Finance et Gardarin 1992]	Relationnel	Best-First sur méta-règles	Langage de règles, Méta-règles
EPOQ [Bjørnestad 1994]	Objet	Planification de tâches	Définition de régions d'optimisation
Extension de SyBase [Andrei et Valduriez 2001]	Relationnel	Parcours de possibilités	Langage de transformation d'opérations, langage d'équivalence d'algorithme
OPT++ [Lanzelotte et Valduriez 1991]	Objet	A implémenter	Langage de définition de stratégie de recherche

TAB. 2.13 – Synthèse des architectures d'optimisation extensible existantes

Ces systèmes d'optimisation extensible permettent d'améliorer les performances du système de gestion de base de données. Cette amélioration dépend des règles de transformation et de la (ou les) stratégie de recherche proposée. Intégré dans un système de médiation de données, un optimiseur extensible facilitera son adaptabilité en fonction de ses besoins. Toutefois, aucun système à ce jour propose des règles de transformation sur des requêtes XQuery, une solution serait d'adapter les optimiseurs extensible orientés objets.

2.4 Conclusion

Nous avons présenté dans ce chapitre différentes architectures de médiation. L'intégration de données semi-structurées pose de nombreux problèmes tels que des structures flexibles, complexes, parfois absentes. De plus, le traitement de requêtes *XQuery* dans un médiateur n'est pas une tâche aisée, compte tenu de la complexité de la grammaire et au fonctionnement de la requête. Une représentation interne adéquate doit s'intégrer au contexte de médiation pour faciliter la localisation des sources de données, les transformations de requêtes, l'intégration de sources et la construction de résultats intermédiaires.

Les principes d'optimisation extensible permettent une meilleure efficacité pour l'optimiseur, orientant l'espace de recherche à l'aide d'une stratégie de recherche adé-

quate. Ainsi, pour des requêtes *XQuery*, l'optimiseur devra intégrer les caractéristiques de ce langage pour permettre des transformations, un modèle interne est donc nécessaire pour ce langage, qui doit faciliter le fonctionnement de l'optimiseur. L'extensibilité de l'optimiseur facilitera l'adaptabilité du système de médiation en modifiant l'optimiseur en fonction des besoins du médiateur.

La problématique de cette thèse est d'intégrer le langage XQuery dans un contexte de médiation. Les techniques d'optimisation extensible doivent permettre l'adaptabilité de l'optimiseur pour améliorer les performances du système de médiation en fonction de ses besoins. Nous proposons donc de définir l'intégralité des traitements d'une requête XQuery dans notre système de médiation *XLive*.

Tout d'abord, un modèle de représentation de requête est nécessaire pour prendre en compte les pré-requis du langage XQuery, et faciliter les étapes d'optimisation et d'évaluation. Nous proposons le modèle *Tree Graph View* (chapitre 3) qui est un modèle graphique complet pour représenter intuitivement les requêtes XQuery.

Un optimiseur est alors nécessaire pour trouver un plan optimal d'évaluation. Dans un souci de configuration de l'optimiseur pour l'adapter en fonction de son environnement, un optimiseur extensible est proposé (chapitre 4). Un langage de définition de règles, une stratégie de recherche et un modèle de coût constituent le cœur de l'optimiseur.

Cette approche originale permet de définir une plate-forme extensible pour le traitement de requêtes XQuery dans un contexte de médiation ; notre modèle de représentation intuitif facilite les manipulations à l'aide de règles de transformations que nous pouvons définir en fonction de l'environnement du système de médiation.

Chapitre 3

Tree Graph View : un modèle de représentation complet pour XQuery

XQuery est un langage de requête riche qui permet d'interroger des documents XML ; ses caractéristiques (Annexe C) nous permettent de catégoriser les éléments principaux du langage. Le but de ce chapitre est de proposer un modèle de représentation de requêtes complet pour *XQuery*. Ce modèle de représentation doit pouvoir intégrer les caractéristiques du langage, Les pré-requis des systèmes de médiation, et faciliter une optimisation à base de règles de transformation. De plus, ce modèle doit pouvoir rester intuitif pour permettre une visualisation de la requête pour aider les utilisateurs.

Le langage *XQuery* comporte de nombreuses caractéristiques qu'il faut intégrer à ce modèle. Nous devons intégrer : les variables de définition, les *XPath*, les imbrications de requêtes, l'optionalité des résultats, les contraintes, les jointures, les projections, les agrégats, les opérations conditionnelles, les opérations ensemblistes, ainsi que les constructions de documents XML.

Afin de faciliter et d'orienter les choix de notre modèle de représentation, une étape de canonisation de requêtes permettra de structurer les requêtes selon un modèle canonique précis. Cette canonisation consiste en une application de règles préservant l'équivalence des requêtes. Le principe de ces règles d'écriture est inspiré par [Chen 2004] que nous avons présenté dans la section 2.2.3. Grâce à ces règles, nous canonisons une requête *XQuery* pour obtenir une structure que nous pouvons exploiter dans notre modèle. Toutefois, les règles proposées par [Chen 2004] ne permettent pas de couvrir l'ensemble des possibilités du langage *XQuery*. En effet, les opérations ensemblistes, les opérations conditionnelles, les séquences, les ordonnancements et les fonctions ne sont pas intégrées dans cette canonisation. Nous proposons alors des règles complémentaires permettant d'enrichir cette étape de canonisation qui nous permet de représenter un ensemble plus conséquent du langage *XQuery*. Une nou-

velle forme canonique sera alors définie pour correspondre aux requêtes canonisées que notre modèle de représentation est capable de couvrir.

Après l'étape de canonisation, la requête canonique est transformée dans notre modèle de représentation. Nous avons étudié dans le chapitre précédent (section 2.2.2) des modèles de représentation de requêtes [Jagadish *et al.* 2001][Amer-Yahia *et al.* 2001][Chen 2004][Arion *et al.* 2005a][Benzaken *et al.* 2004] et nous avons pu constater que chacun avait ses propres avantages et inconvénients. Deux approches ont plus particulièrement inspiré notre modèle de représentation :
- les *Tree Pattern Queries* [Jagadish *et al.* 2001] ou *TPQ* (motif d'arbre) permettent de représenter un filtre sur un document XML unique, intégrant les principaux problèmes liés à *XQuery* (liens hiérarchiques, optionnalité, contraintes) ;
- les *Generalized Tree Pattern* [Chen 2004] sont une généralisation des *TPQ*, qui couvrent un plus grand ensemble du langage de requêtes. En effet, ils permettent de reconnaître les jointures entre collections, les imbrications, et les quantificateurs.

Ces deux représentations fournissent une représentation interne des requêtes. Elles permettent d'identifier les domaines de définition à interroger grâce à leurs variables (déclarées dans une clause *for*). Toutefois, celles-ci manquent en expressivité :

- la notion de sources de données n'est pas incluse ;
- le résultat XML d'une requête n'est pas modélisé ;
- modéliser des vues et des requêtes sur des vues n'est pas possible ;
- les LET et les fonctions ne sont pas représentés ;
- les balises, les relations, et les contraintes sont incluses dans une formule booléenne difficile à lire, ayant un lien non intuitif avec la représentation ;
- ni les opérations ensemblistes, ni les opérations conditionnelles ne peuvent être ajoutées.

Nous proposons un modèle de représentation de requêtes pour *XQuery* permettant de couvrir un plus grand ensemble du langage et permettant de faciliter les problèmes d'optimisation que nous allons étudier dans le chapitre 4. De plus, la réflexion a été orientée pour faciliter les pré-requis d'un système de médiation, comme l'identification des sources, la modélisation des contraintes et des capacités ou la structure des résultats temporaires. Ce modèle de représentation se base sur les principes des *Generalized Tree Pattern* [Chen 2004] avec des contraintes applicables à chaque *Pattern*, des liens reliant les *TreePattern*, et un *TreePattern* de reconstruction de résultat. Nous l'appellons *Tree Graph View* (TGV), une vue des requêtes XQuery sous forme de graphes de motifs d'arbre.

Ce chapitre est organisé comme suit. Pour commencer, nous développons un exemple détaillant chacune des étapes de la traduction : *XQuery*, *XQuery canonique* et *TGV* (section 3.1). Suivant les étapes données dans cet exemple, nous étudions ensuite l'étape de canonisation de requêtes (section 3.2) qui permet de préparer la modélisation sous forme de *TGV*. Nous définissons alors ce modèle de représentation (section 3.3) pour ensuite les formaliser par l'intermédiaire des *Types Abstraits de Données* ou *ADT* (*Abstract Data Type*) (section 3.4). Pour finir, nous proposons ensuite un algorithme de traduction du langage *XQuery* en *TGV* (section 3.5).

3.1 Traitement de requêtes XQuery

Dans un processeur de requêtes, nous pouvons distinguer la partie traduction et la partie évaluation. La traduction des requêtes XQuery débute par la canonisation pour préparer la requête, puis par la modélisation en TGV. L'évaluation des requêtes XQuery sera étudiée dans le chapitre suivant. Comme le montre la figure 3.1, le traitement de la requête XQuery se décompose en trois étapes. Ses trois étapes sont développées tout au long de ce chapitre. La première prend une requête XQuery non typée (voir section 3.2.6), car le typage n'est pas reconnu par le modèle TGV. Cette requête non typée est alors transformée à l'aide des lemmes de canonisation de requêtes (section 2.2.3 et section 3.2), la requête obtenue est alors dite *canonique*. A partir d'une requête canonique, les algorithmes de traduction (section 3.5) sont alors appliqués pour transformer la requête dans le modèle TGV (section 3.3), sous forme de types abstraits de données (section 3.4). A l'aide des TGV, nous sommes capable de représenter l'ensemble des requêtes canoniques non typées soumises au processeur de requête.

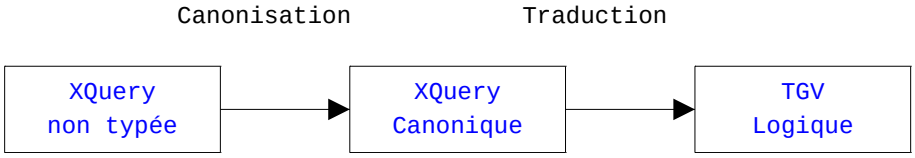


FIG. 3.1 – Cycle d'évaluation d'une requête XQuery

XQuery non typée

Pour illustrer le processus de traitement de requête XQuery, nous appliquons celui-ci sur un exemple. La requête du tableau 3.1, demande l'ensemble des éléments *book* des collections «*reviews*» et «*catalogs*» devant contenir le mot «*daulphin*» et dont le prix doit être supérieur à 15. Enfin, pour chacun de ces livres, son identifiant et son prix est affiché, et son titre est récupéré à partir de la collection «*books*».

```
for $x in ( collection("reviews")/review | collection ("catalogs")/catalog )
            [. contains("daulphin")]/book/[.//price > 15]
return <livre>
            {$x/@isbn}
            <prix>{$x//price/text()}</prix>
            {for $z in collection("books")/book
            where $z/@isbn = $x/@isbn
            return <titre>{$z/title/text()}</titre>}
        </livre>
```

TAB. 3.1 – Exemple de requête XQuery non typée

Pour détailler un peu plus la requête, les deux ensembles «*reviews*» et «*catalogs*», auquels nous associons un path, sont unis (opérateur «|») pour donner la variable $x

sur cet ensemble nous cherchons ceux qui contiennent le mot «*dauphin*» (fonction *contains*), puis pour ceux qui ont le path «*book*» nous voulons ceux dont l'élément «*price*» vaut plus de 15 (filtre *price > 15*). Pour cette déclaration de *for* nous créons un document XML, dont on aura l'attribut «*@isbn*» et son prix. De plus, nous imbriquons une requête pour récupérer le titre des livres (collection «*book*») ayant le même «*@isbn*» (prédicat de jointure entre $z et $x).

XQuery canonique

Grâce aux spécifications du W3C, une requête XQuery peut être écrite de différentes manières. Pour éviter cette multitude d'expressions possible, une étape de canonisation permet de transformer une requête dans une forme canonique, plus simple à traduire et à modéliser. La section 3.2 développe cette étape. Le tableau 3.2 représente la requête XQuery canonisée à partir de la requête du tableau 3.1. Tout d'abord, les filtres ont été détachés de la clause *for* pour les associer à la clause *where* lui correspondant (fonction *contains* et prédicat sur le prix). Une nouvelle variable $y a été insérée pour représenter le filtre *contains (., "daulphin")* au milieu du *XPath*. Une clause *let* l_1$ est déclarée pour reprendre la requête imbriquée, la variable étant projetée dans la clause *return*. Concernant l'union des deux collections, une clause *let* a été ajoutée créant un ensemble pour chacun et l'opération ensembliste dans la clause *return*.

```
let $u3 := (for $u1 in collection ("reviews")/review
            for $u2 in collection ("catalogs")/catalog
            return ( $u1 | $u2 ) )
for $x in $u3, $y in $x/book
Let $l1 := (for $z in collection ("books")/book
            where $z/@isbn = $y/@isbn
            return <titre> {$z/title/text()} </titre>)
where contains ($x, "daulphin") and $y//price > 15
return <livre>
            {$y/@isbn}
            <prix>{$y//price/text()}</prix>
            {$l1}
        </livre>
```

Tab. 3.2 – XQuery canonique généré à partir de l'exemple 3.1

Transformation et Représentation

La dernière étape du traitement des requêtes XQuery est la transformation d'une requête canonique dans le modèle TGV. Ce modèle est développé dans la section 3.3. La figure 3.2 illustre la représentation sous forme graphique des TGV obtenue par transformation de la requête canonique du tableau 3.2. Les algorithmes de transformations ont été utilisés pour transformer la requête (section 3.5. Nous pouvons identifier les deux domaines «*reviews*» et «*catalogs*» avec l'opération *union* en dessous («*initSetOperator*») qui se projette dans un *STP* («*initFor*»). Ce dernier possède une opération *contains* («*initWhere*»), et un hyperlien d'exploration du nœud *book* vers l'*ITP* $y («*initFor*») dont le prix doit être supérieur à 15 («*initWhere*»). Sur le coté droit, nous pouvons identifier la requête imbriquée grâce à l'*ATP* «*return*» («*initLet*») où se projette le titre du second *STP* sur la collection «*books*». Les deux ensembles sont reliés par un *JoinHyperlink* entre leurs attributs «*@isbn*» («*initFor*»). Pour finir, le tout est projeté vers le *ReturnTreePattern* dont la racine est la balise «*livre*» sur laquelle se projette l'attribut *isbn*, le prix et le titre («*initReturn*»).

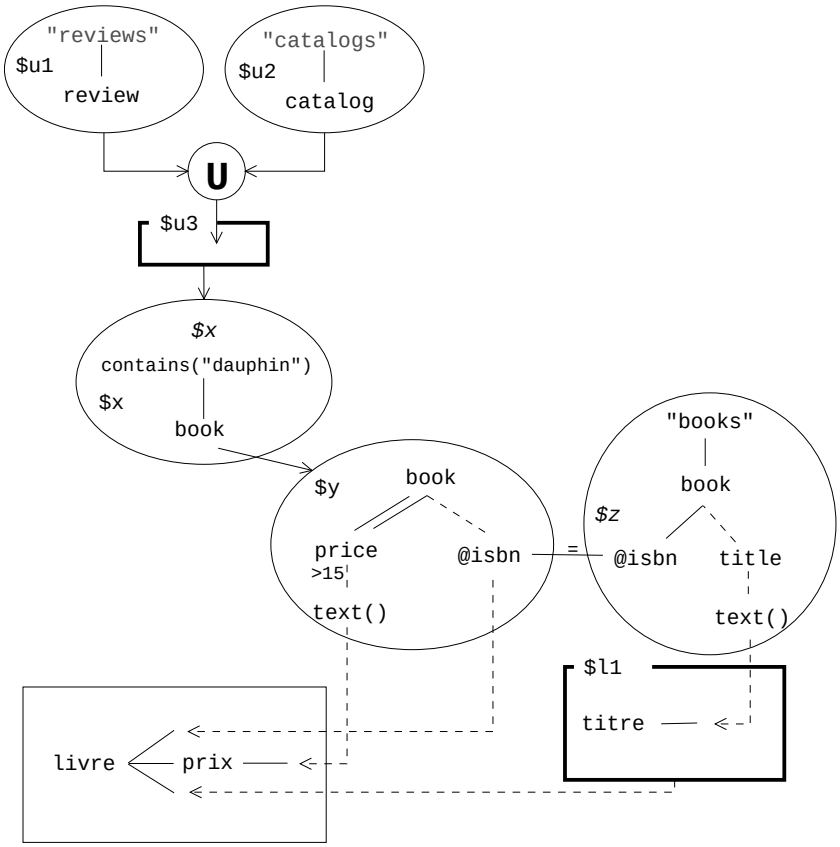


FIG. 3.2 – TGV de la requête du tableau 3.1

Nous avons pu voir dans cette section qu'il était possible de transformer une requête XQuery en une forme canonique. Celle-ci peut alors être traduite en *TGV* pour obte-

nir une représentation logique et simple des requêtes XQuery. Le langage de requête XQuery peut alors être traduit sous cette forme pour nous permettre d'identifier les domaines caractéristiques (ou documents cibles avec leurs filtres associés) et de les manipuler à l'aide d'opérations. Nous développons à présent chacune de ces étapes.

3.2 Canonisation de XQuery

Comme nous avons pu le voir, les règles de transformation de requêtes de [Chen 2004] nous permettent d'obtenir une requête *XQuery* canonisée. Cette étape de canonisation permet de formater les requêtes en une forme canonique plus aisément modélisable. Toutefois, celles-ci sont insuffisantes pour couvrir entièrement ce langage. Nous proposons donc une extension de ces règles pour nous permettre de recouvrir un plus grand ensemble pour le langage de requêtes *XQuery*, mais aussi pour nous permettre de les manipuler plus facilement.

Dans [Chen 2004], quatre familles d'expression sont manquantes à la canonisation : l'ordonnancement, les opérations ensemblistes, les opérations conditionnelles et les séquences. Chacune de ces expressions permettent de représenter des manipulations d'ensembles d'arbres. Nous proposons donc pour chacune d'elles une canonisation qui pourra être ajoutée à l'ensemble des règles. Comme nous le voyons par la suite, certaines expressions n'ont pas de correspondance dans la grammaire *XQuery* canonique définie par [Chen 2004], nous proposons alors une nouvelle grammaire permettant d'intégrer ces expressions.

3.2.1 Ordonnancement

L'ordonnancement est une opération permettant de classer les arbres XML selon un critère donné par un ou plusieurs *XPath*. L'ordre des arbres est donné par le classement des nœuds par valeur, provenant de cet *XPath*. Cette opération prend un ensemble d'arbres et produit un nouvel ensemble ordonné.

Pour obtenir une requête *XQuery* canonique, la clause *orderby* doit être transformée en une clause reconnue par l'expression canonique. Si l'on considère l'opération d'ordonnancement, celle-ci est appliquée après les calculs des clauses *for*, *let* et *where*, mais avant la clause *return*. Ainsi, le résultat des opérations précédant la clause *orderby* peut être traité par une fonction de tri et associé à une clause *let* comme une fonction d'agrégat.

Lemme 3.1 : Ordonnancement

Une requête *XQuery* contenant une clause *OrderBy* peut être transformée en une requête équivalente sans cette clause. Elle est déclarée dans une clause *let* avec une fonction *orderby*() dont les paramètres d'entrées sont les domaines à ordonner, définis par leurs variables, ainsi que les *XPath* de tri et l'ordre du tri. Le résultat de cette fonction est l'ensemble d'arbres triés à partir de l'ensemble de départ, la nouvelle variable remplace les variables utilisant le domaine d'origine. Pour les variables définies par une clause *for*, une nouvelle variable est utilisée pour reproduire le flux d'arbres.

Preuve : Soit une requête Q. Si Q ne contient pas de clause *orderby*, elle est alors canonique (pour les critères d'ordonnancement).

Supposons que Q possède n clauses *orderby* de type : «*order by* $\$var_1/path_1$, ... $\$var_n/path_n$». Grâce aux lemmes de transformations de chemins, les XPaths $path_x$ sont canoniques. La requête Q est considérée comme canonique si la clause *orderby* est remplacée par une clause *let* définissant l'ordonnancement, et chacune des variables correspondantes transformées.

Il faut alors traiter 3 cas de clause *orderby* :

1. Si une variable est déclarée : «*order by* $\$var_1/path_1$ *return* $\$var_1/path_2$», alors : «*let* $\$t$:= *orderby* ($\$var_1$, $\$var_1/path_1$) *return* $\$t/path_2$» ;
2. Si deux variables (ou plus) sont déclarées, mais identiques : «*order by* $\$var_1/path_1$, $\$var_1/path_2$ *return* $\$var_1/path_3$», alors : «*let* $\$t$:= *orderby* ($\$var_1$, $\$var_1/path_1$, $\$var_1/path_2$) *return* $\$t/path_3$» ;
3. Si deux variables (ou plus) sont déclarées, mais différentes : «*order by* $\$var_1/path_1$, $\$var_2/path_2$ *return* $\{\$var_1/path_3, \$var_2/path_4\}$», alors : «*let* $\$t_1$:= *orderby* ($\$var_1$, $\$var_1/path_1$), $\$t_2$:= *orderby* ($\$var_2$, $\$var_2/path_2$) *return* $\{\$t_1/path_3, \$t_2/path_4\}$».

Pour reproduire l'évaluation des arbres, une nouvelle variable est déclarée dans la clause *for* en utilisant le résultat de l'agrégat de la clause *let*. Ainsi, les arbres triés sont récupérés l'un après l'autre.

Alors, si pour la $n+1$ ème expression *orderby* de la requête Q peut être écrite avec n expression *orderby*, puisqu'une requête ne contenant pas d'expression *orderby* est canonique, alors par récursivité Q peut être écrite sans clause *orderby*.

Voici un exemple de canonisation de clause *orderby* :

Requête XQuery	**Requête XQuery Canonique**
for $i in /catalog/book order by $i/title return $i/title	for $i in /catalog/book let $j := orderby ($i, $i/title) for $k in $j return $k/title

TAB. 3.3 – Exemple de canonisation de la clause *orderby*

Dans l'exemple du tableau 3.3, la clause *for* sélectionne l'ensemble des éléments *book* contenus dans *catalog*. Puis, l'ensemble est trié sur les valeurs de l'élément *title*, puis ceux-ci sont associés à la variable $j. La canonisation de la clause *orderby* donne une clause *let* : $j, dont la fonction d'ordonnancement *orderby*() prend la variable $i en entrée pour le domaine, et l'*XPath* $/title pour tri. Le résultat est alors réutilisé dans une clause *for* ($k) pour retrouver l'évaluation en flux des arbres de la variable $i. Le tout est utilisé dans la clause *return* lors du retour du titre avec l'*XPath* $k/title (au lieu de $i/title). Nous avons alors une requête canonisée ne contenant plus de clause *orderby* mais une clause *let* contenant la fonction de tri *orderby*.

3.2.2 Opérations Ensemblistes

Les opérations ensemblistes permettent de faire des unions, des différences ou des intersections d'ensembles d'arbres. L'opération traite deux, voire plusieurs, ensembles d'arbres pour produire un ensemble unique. L'opération *union* permet de rassembler tous les ensembles d'arbres, la *différence* permet de supprimer du premier ensemble d'arbres, ceux présents dans le second, alors que l'*intersection* permet de donner l'ensemble des arbres présents dans les deux ensembles donnés. Cette opération traite donc plusieurs ensembles d'arbres.

Lemme 3.2 : Opération Ensembliste

Une requête XQuery contenant une clause ensembliste peut être transformée en une requête équivalente où l'expression est décomposée et contient une clause let avec les deux expressions et la clause *return* contient l'opération ensembliste entre les deux expressions.

Preuve : Soit une requête Q. Si la requête Q ne contient pas d'opération ensembliste entre deux expressions FLWR, alors elle est dite canonique.

Si la requête Q contient $n+1$ opérations ensemblistes entre deux expressions (autres que des variables), grâce aux lemmes de canonisation, les deux expressions sont canoniques. Soit ξ l'opération ensembliste définie dans l'ensemble de valeur {"*union*", "*intersect*", "*except*"} (union, intersection, différence). Le tableau 3.4 illustre les quatre transformations possibles :

Ainsi, une requête Q contenant $n+1$ opérations ensemblistes entre deux expressions peut être écrite avec n opération ensemblistes. S'il n'y a pas d'opération ensembliste, elle est canonique. Alors, par récursivité, toute requête Q peut être canonisée sans opération ensembliste.

Voici un exemple de canonisation d'une clause ensembliste :

Expression Ensembliste	Expression Canonisée	Explications
$(expr_1\ \xi\ expr_2)$	let $\$t_3$:= for $\$t_1$ in $expr_1$ for $\$t_2$ in $expr_2$ return ($\$t_1\ \xi\ \$t_2$)	Chaque expression est définie par une nouvelle variable. Celles-ci sont reliés par l'opérateur.
$(expr_1\ \xi\ expr_2)/P$	let $\$t_3$:= for $\$t_1$ in $expr_1$ for $\$t_3$ in $expr_2$ return ($\$t_1\ \xi\ \$t_2$) ... $\$t_3/P$	L'expression est décomposée. 1) l'opérateur ensembliste 2) le chemin remplacé par la variable.
$\$XP(P_1\ \xi\ P_2)$	for $\$t_x$ in XP let $\$t_3$:= for $\$t_1$ in $\$t_x/P_1$ for $\$t_2$ in $\$t_x/P_2$ return ($\$t_1\ \xi\ \$t_2$)	Une nouvelle variable est créée. L'opération ensembliste s'applique sur la nouvelle variable
$\$XP(P_1\ \xi\ P_2)/P_3$	for $\$t_x$ in XP let $\$t_3$:= for $\$t_1$ in $\$t_x/P_1$ for $\$t_2$ in $\$t_2/P_2$ return ($\$t_1\ \xi\ \$t_2$) ... $\$t_3/P$	La composition des règles 2 et 3 produit la décomposition de l'expression ensembliste comprise entre XP et P_3

TAB. 3.4 – Transformations des opérations ensemblistes

Requête XQuery	Requête XQuery Canonique
for $i in (/catalog | /review)/book return $i/title	let i_3$:= for $\$i_1$ in /catalog for $\$i_2$ in /review return (i_1$ | i_2$) for $i in i_3$/book return $i/title

TAB. 3.5 – Exemple de canonisation d'une opération ensembliste

Dans l'exemple du tableau 3.5, la clause for contient l'union «|» des ensembles */catalog* et */review* sur lesquels est sélectionné l'élément *book*. Le titre est alors projeté pour chacun de ces livres. La canonisation de l'opération d'union (raccourci «|») donne une clause *let* ($\$i_3$) contenant deux expressions i_1$ et i_2$ sur les xpath requis pour l'union des deux ensembles. la clause *let* retourne alors l'union des deux expressions. L'opération est contenue dans $\$i_3$ en utilisant les variables définies précédemment. Enfin, le résultat est utilisé dans la clause for avec l'*XPath* : $\$i_3/book$. Nous obtenons alors une requête canonisée où les opérations ensemblistes sont décomposées pour détailler le mode opératoire de chacune des expressions reliées par l'opérateur.

3.2.3 Opérations Conditionnelles

Les opérations conditionnelles donnent une notion opérationnelle du traitement de résultats. En effet, le résultat de la condition dépend de la contrainte imposée à l'opération. Deux résultats sont possibles, si la contrainte est vérifiée un premier résultat est donné, sinon le second est retourné. Ainsi, le résultat peut être une requête imbriquée ou un *XPath*, il est alors nécessaire de désimbriquer les requêtes.

Lemme 3.3 : Opération Conditionnelle

Une requête XQuery contenant un opérateur conditionnel (if/then/else) et une requête imbriquée dans celle-ci peut être transformée en une requête équivalente où la requête imbriquée sera déclarée dans une clause *let*.

Nous pouvons démontrer ce lemme de la même manière que le lemme de désimbrication de [Chen 2004] (section 2.2.3) concernant les requêtes imbriquées d'une clause *return*. Ainsi, par récursivité, nous prouvons que toute requête contenant une expression imbriquée dans une opération conditionnelle pourra être canonisée.

Voici un exemple de canonisation de requête avec un opérateur conditionnel dont le résultat est une expression imbriquée :

Requête XQuery	**Requête XQuery Canonique**
for $i in /catalog/book return {if contains ($i/author, "Hobb") then (for $j in $i//title return $j) else ($i/author)}	for $i in /catalog/book let $l := for $j in $i//title return $j return {if contains ($i/author, "Hobb") then ($l) else ($i/author)}

Tab. 3.6 – Exemple de canonisation d'opération conditionnelle

Dans l'exemple du tableau 3.6, une opération conditionnelle est déclarée dans la clause *return* avec pour contrainte le nom de l'auteur qui doit contenir le mot «*Hobb*». S'il contient ce mot, la requête imbriquée $j doit retourner les titres de ce livre, sinon, l'auteur est retourné. La canonisation de l'opération conditionnelle déplace les requêtes imbriquées dans les conditions de retour pour les déclarer dans une clause *let*. La nouvelle variable $l remplace la requête dans la clause conditionnelle pour déclarer un résultat canonisé.

3.2.4 Séquences

Les séquences sont des groupements d'ensembles d'éléments sur lesquels sont appliqués des opérations. En effet, lorsqu'une contrainte est appliquée sur une séquence à l'aide de parenthèses "(" *XPath* ")", la contrainte est appliquée sur l'ensemble des arbres désignés par l'XPath (et non sur chacun de ces arbres). Cette opération regroupe des ensembles d'arbres pour ensuite appliquer la contrainte sur celui-ci.

Lemme 3.4 : Séquences

Une requête XQuery contenant une séquence peut être écrite en une requête équivalente sans séquences. Chaque séquence est traduite dans une clause *let* qui pourra alors être utilisée.

Les séquences se comportent de la même manière qu'un filtre à l'intérieur d'un XPath. Nous démontrons alors de la même manière que le lemme de canonisation des filtres de [Chen 2004]. Ainsi toute expression d'une séquence sera déclarée dans une clause *let*, générant ainsi une nouvelle variable utilisée pour le reste du chemin.

Requête XQuery	**Requête XQuery Canonique**
for $i in (/catalog/book)[2] return $i/title	let i_1$:= for $x in /catalog/book return $x for $i in i_1$ where $i/position() == 2 return $i/title

TAB. 3.7 – Exemple de canonisation des séquences

Dans l'exemple du tableau 3.7, une séquence est définie dans l'*XPath* de la clause *for*, l'ensemble des livres des catalogues sont agrégés pour ensuite ne sélectionner que le second (et non le second livre de chaque catalogue) puis, le titre de ce livre est projeté. L'étape de canonisation a produit une clause *let* dans laquelle est déclarée la clause *for* requise par la séquence. Ensuite, la nouvelle variable est utilisée pour l'opération et pour la clause *return*.

3.2.5 Fonctions

Les définitions de fonctions sont utiles pour définir une requête qui pourra être réutilisée de nombreuses fois, ou pour définir des requêtes avec des paramètres variables. Dans le langage *XQuery*, la fonction prend des paramètres en entrée et un ensemble unique en sortie, les entrées et la sortie sont typés. Dans le cadre de notre travail, nous ne considérons comme typage que les éléments, les booléens et les nombres, en effet, ceux-ci recouvrent un large ensemble de types de valeurs. L'étape de canonisation des fonctions ne s'occupe alors que de canoniser la requête définie dans la fonction.

Lemme 3.5 : Fonctions

Une requête XQuery contenant une fonction avec une expression *XQuery* peut être écrite en une requête équivalente contenant une fonction avec une expression canonique.

Dans l'exemple du tableau 3.8, une fonction est définie (*local :section*) avec un paramètre en entrée. Cette entrée est définie par la clause *for* : *for $f in collection("catalog")/catalog*, qui sera utilisée dans l'appel de fonction : *local :section($f)*. Ce domaine de valeur est utilisé pour chaque élément *book* pour définir son titre,

et l'ensemble de tous les titres de sections (*$/section/title*) qu'il possède. Comme nous pouvons le constater, la fonction contient une requête imbriquée, l'étape de canonisation désimbrique la requête pour obtenir une requête canonisée à l'intérieur de la fonction.

Requête XQuery	**Requête XQuery Canonique**

declare function local :section
($i as element()) as element ()*
{
for $j in $i/book
return
<book>
{$j/title}
{for $s in $i/section/title
return <section>
{$s/text()}
</section>}
</book>
}
for $f in doc("catalog.xml")/catalog
return local:section($f)
	declare function local :section
	($i as element()) as element ()*
	{
	for $j in $i/book
	let $l := (for $s in $i/section/title
	return <section>
	{$s/text()}
	</section>)
	return
	<book> {$j/title} {$l} </book>
	}
	for $f in doc("catalog.xml")/catalog
	return local:section($f)

TAB. 3.8 – Transformation d'une fonction

3.2.6 XQuery Canonique

Ainsi grâce aux lemmes précédents associés à ceux proposés par [Chen 2004], nous pouvons reconnaître un large ensemble d'expressions du langage *XQuery*. Nous pouvons reconnaître à présent :
- *les expressions XPath contenant des filtres* ;
- *les clauses for, let et return* ;
- *les prédicats de la clause where* ;
- *les imbrications de requêtes* ;
- *les fonctions d'agrégation* ;
- *les quantificateurs* ;
- *les ordonnancements* ;
- *les opérations ensemblistes* ;
- *les opérations conditionnelles* ;
- *les séquences* ;
- *les définitions de fonctions* ;

Les typages ne sont pas intégrés à cette étape de canonisation. En effet, dans notre contexte de médiation, les typages n'ont pas été considérés et ne sont représentés dans les TGV. De ce fait, l'étape de canonisation de prend pas en compte le typage des éléments.

Théorème 3.1 : Canonisation

Toute requête *XQuery* non typée peut être canonisée.

La grammaire *XQuery* canonique présentée dans la définition 3.2.2.3 correspond à la canonisation de la grammaire donnée en annexe C. Nous pouvons observer que toute requête *XQuery* canonique commence par une requête FLWR accompagnée d'une ou plusieurs définitions de fonctions. De plus, une différence est visible sur la définition des expressions *Expr* décomposée avec une expression canonique *CanonicExpr*, permettant de différencier les imbrications et les agrégats, des chemins et des fonctions non agrégatives. De plus, les opérations ensemblistes sont intégrées dans ces expressions, tandis que les opérations conditionnelles sont intégrées à la clause de reconstruction de résultats *ReturnClause*. Le champ *Declaration* a aussi une forme canonique interdisant les imbrications d'expressions dans celui-ci. Les chemins *XPath* ne comportent plus de filtres, de séquences, ni d'opérations ensemblistes puisque ceux-ci sont canonisés.

Définition 3.1 : XQuery Canonique non typée

XQuery : :=	(*Function*)* FLWR ;
FLWR : :=	("for" "$" STRING " in " *Declaration*
	(, "$" STRING " in " *Declaration*)*
	| "let" "$" STRING " : :=" "(" *Expr* ")"
	(, "$" STRING " : :=" "(" *Expr* ")")*
	)+
	("where" *Predicate* (("and" | "or") *Predicate*)*) ?
	"return " *ReturnClause* ;
ReturnClause : :=	"{" *CanonicExpr* "}"
	| "{" "if" *Predicate* "then" "(" *Expr* ")" "else" "(" *Expr* ")" "}"
	| "<" STRING ">" (*ReturnClause*)* "</" STRING ">" ;
Expr : :=	*FLWR* | "(" *Path SetOperator Path* ")"
	| *CanonicExpr* | *aggregate_function* ;
CanonicExpr : :=	*Path* | *non_aggregate_function* ;
Declaration : :=	"collection" "(' " STRING " ')" (*XPath*) ? | *CanonicExpr* ;
Path : :=	"$" STRING *XPath* (EndXPath) ? ;
Predicate : :=	*Val Comp Val* | *QName* "(" ((*Val* ",")* *Val*) ? ")" ;
Comp : :=	">" | "<" | "=" | "<=" | ">=" | "! =" ;
Val : :=	' STRING ' | Number | *Path* ;
XPath : :=	("/" *Element* | "//" *Element*)+ ;
SetOperator : :=	"|" | "-" | "/" ;
EndXPath : :=	"/" (*Attribut* | *Element* | "text()") ;
Element : :=	(*QName* | "." | "..");
Attribute : :=	"@" *QName* ;
QName : :=	(STRING " :") ? STRING ;
Function : :=	"declare function" QName "(" "$" STRING "as" "element"
	("," "$" STRING "as" "element")* ")" "as" "element"
	"{" FLWR "}" ;

Le tableau 3.9 récapitule les règles de canonisation d'une requête *XQuery* proposées dans cette section. Associées aux règles données dans la section 2.2.3, elles permettent de définir une canonisation de toutes les requêtes XQuery non typées.

	Expressions		**Forme canonique**
R1	*order by var/xp*	$\Rightarrow$	*let* $\$l_1 := orderby(var, var/xp)$
R2	($expr_1$ union $expr_2$)	$\Rightarrow$	*let* $\$i_3$:= for $\$i_1$ in $expr_1$, $\$i_2$ in $expr_2$ return ($\$i_1$ union $\$i_2$)
	($expr_1$ intersect $expr_2$)	$\Rightarrow$	*let* $\$i_3$:= for $\$i_1$ in $expr_1$, $\$i_2$ in $expr_2$ return ($\$i_1$ intersect $\$i_2$)
	($expr_1$ except $expr_2$)	$\Rightarrow$	*let* $\$i_3$:= for $\$i_1$ in $expr_1$, $\$i_2$ in $expr_2$ return ($\$i_1$ except $\$i_2$)
R3	*if* $expr_1$		*let* $\$l_1 := expr_2, \$l_2 := expr_3$
	then $expr_2$	$\Rightarrow$	*if* $expr_1$ *then* $\$l_1$ *else* $\$l_2$
	else $expr_3$		(si $expr_2$ et $expr_3$ sont des requêtes imbriquées)
R4	$(expr_1)/expr_2$	$\Rightarrow$	*let* $\$l_1 := expr_1$ $\$l_1/expr_2$

TAB. 3.9 – Récapitulatif des règles de canonisation

3.3 Définition du modèle

La modélisation des requêtes *XQuery* est une tâche complexe compte tenu de la richesse de ce langage. Comme nous avons pu le voir précédemment, *XQuery* produit un grand nombre d'expressions que nous devons intégrer au modèle. L'étape de canonisation, que nous avons étudié dans les sections 2.2.3 et 3.2, nous permet d'obtenir une forme canonique que nous pouvons utiliser pour la modélisation.

Nous avons pu voir les différentes modélisations de requêtes semi-structurées dans la section 2.2.2 [Jagadish *et al.* 2001][Amer-Yahia *et al.* 2001][Chen 2004] [Arion *et al.* 2005a]. Dans un contexte de médiation, nous devons intégrer des informations supplémentaires qui ont orienté la spécification de nos motifs d'arbres appelés *Tree Pattern* (section 3.3.1). Les nuances entre les différents motifs possibles (*for*, *let*, *return*) ont donné une spécification de chacun de ces modèles. De plus, les *contraintes* (section 3.3.2) que nous pouvons déclarer dans les différentes expressions nous a permit de distinguer motifs d'arbre et contraintes externes. Enfin, les liens unissant ces différents motifs ont orienté notre réflexion sur les *hyperliens* (section 3.3.3), liens externes aux motifs d'arbre.

3.3.1 Motifs d'Arbres

Dans une requête *XQuery*, nous pouvons déclarer des ensembles d'arbres grâce aux mots clés *for* et *let*, auxquels nous associons une variable que nous nommons une variable de définition, permettant de déterminer un domaine de valeur. A chacune

de ces variables peut être associé un ou plusieurs chemins (ou *XPath*) produisant une structure arborescente d'éléments qu'un document XML doit respecter pour être sélectionné. Nous pouvons identifier ces éléments par des nœuds correspondant aux noms des balises. Les liens reliant deux nœuds déterminent les axes de recherches (développé en annexe B) entre ces deux éléments. Enfin, la notion d'optionalité (étudié dans l'annexe C) d'un élément provient des chemins déclarés dans la clause *return* d'une requête, ainsi la liaison entre deux nœuds peut être optionnelle.

Ainsi, un motif d'arbre (ou *TreePattern*) représente les différentes balises qu'un document doit contenir pour être sélectionné. Il contient une hiérarchie de nœuds donnant la structure arborescente du document requis. Chaque chemin d'une requête *XQuery* génère un ensemble de *nœuds* et de *liens entre nœuds* lié à une variable de définition.

Définition 3.2 : Nœud (Node)

> Un *nœud* est un élément d'un motif d'arbre. Il défini la correspondance entre un élément du document XML et le nom du nœud. Cette correspondance dépend du type de motif d'arbre du nœud.

Un nœud défini la correspondance entre un élément d'un document XML et le nom de l'élement du motif d'arbre. Chaque nœud d'un *TreePattern* doit être présent dans le document ciblé. Dans le cas des motifs d'arbre source et intermédiaire, chaque nœud filtre l'élément ciblé. Dans le cas des motifs d'arbre résultat ou d'agrégation, les nœuds correspondent à la création d'un élément dans le document XML. Afin de représenter les spécificités de *XQuery*, les *namespace* et les *astérisques* sont intégrés aux noms des nœuds.

Définition 3.3 : Lien de nœud (NodeLink)

> Un *lien de nœud* est un axe de parcours entre deux nœuds. Il contient trois informations : axe de parcours entre les deux nœuds, optionalité du lien (obligatoire ou optionnel) et l'indice de déclaration dans la requête.

Un lien de nœud représente une relation entre deux nœuds. Ainsi, deux éléments d'un document XML doivent respecter cette relation pour être validé par le motif d'arbre. L'association entre les deux nœuds dépend de l'axe de parcours du document. Seuls les axes *child*, *descendant*, *descendant-or-self*, *ancestor*, *following-sibling*, *preceding-sibling*, *following*, *preceding*, et *ancestor-or-self* sont utilisés des liens (les autres sont simplifiés et considérés comme des nœuds).

Concernant l'information d'optionalité d'un nœud, le second élément du lien doit apparaître si cette information est obligatoire. Tandis que si cette information est optionnelle, alors l'arbre XML n'est pas éliminé si cette information est absente.

L'indice de déclaration correspond à l'ordre de déclaration des *XPath* dans une

requête *XQuery*, cet indice implique un ordre de vérification[1], et non un ordre d'apparition dans le document XML (cette information correspondant à une contrainte).

Représentation graphique des nœuds et des liens de nœuds

Pour représenter graphiquement cet ensemble de nœuds, chaque nœud est modélisé par son label, chaque *NodeLink* reliant deux nœuds est représenté sous forme d'un lien ; (1) plein lorsqu'il est obligatoire, (2) en pointillés lorsqu'il est optionnel, (3) simple lorsque l'axe entre les deux nœuds est de type '*child*', et (4) double lorsqu'elle est de type '*descendant*', ces deux représentations étant les plus courantes en *XQuery* sont utilisés tel quel. Concernant les autres axes possibles, les liens sont annotés par leur nom respectif :
- *descendant-or-self* : lien double avec mot-clé **or-self** ;
- *ancestor* : un lien double avec mot-clé **a** ;
- *ancestor-or-self* : lien double avec mot-clé **a-or-self** ;
- *following* : lien avec mot-clé **f** ;
- *following-sibling* : lien avec mot-clé **fs** ;
- *preceding* : lien avec mot-clé **p** ;
- *preceding-sibling* : lien avec mot-clé **ps** ;

L'ordre des nœuds définit la position du lien lors de sa déclaration dans la requête, il n'influe pas sur sa position dans le document (induit par une contrainte ou un lien).

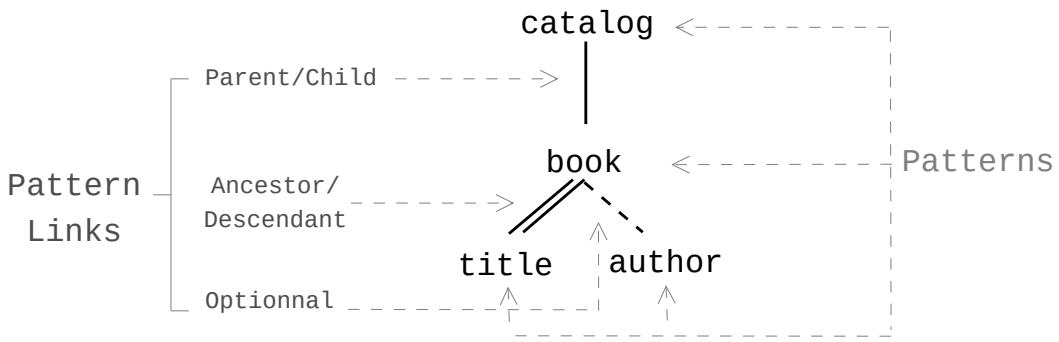


FIG. 3.3 – Ensemble de nœuds

Nous pouvons ainsi représenter les nœuds «*catalog*», «*book*», «*title*» et «*author*» à l'aide de *NodeLink*s interconnectant cet ensemble de nœuds. La représentation graphique de cet ensemble illustré par la figure 3.3 contient trois *NodeLink*s. Le premier entre *catalog* et *book* est obligatoire (trait plein). Le second lien entre *book* et *title* est obligatoire, en première position et la relation est de type ancêtre/descendant, alors que le troisième lien entre *book* et *author* est optionnel, de type parent/enfant et en seconde position.

La figure 3.4 illustre un motif d'arbre (à gauche) que l'on applique sur un document XML (à droite). Ce document est un catalogue contenant deux livres («l'assassin royal» et «la nef du crépuscule»), tous deux ont un titre, par contre seulement un

[1]Cette information peut avoir son importance dans l'étape d'évaluation.

seul possède un auteur. Chaque livre contient une ou plusieurs sections avec un titre. Le motif d'arbre sélectionne tous les titres quelque soit son ascendance (double lien) et un auteur optionnel. Ainsi, la réponse à ce motif d'arbre correspond à la sélection des deux livres, avec tous les titres, y compris ceux des sections, et l'auteur s'il existe (premier livre).

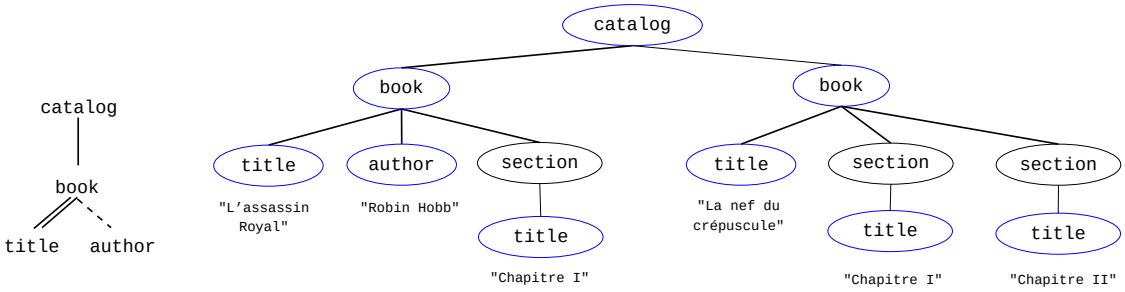


FIG. 3.4 – Motif d'arbre appliqué à un document XML

Ainsi, grâce aux nœuds et liens de nœuds nous pouvons former un motif d'arbre (*TreePattern*) que nous pouvons utiliser afin de définir un filtre pour document XML. Un domaine est déterminé par sa variable de définition et une racine de document (chemin de déclaration d'une clause *for*), cette racine détermine le sous-ensemble du document XML qui est défini par la variable de définition. Chaque chemin associé à cette variable est lié à la racine produisant ainsi le motif d'arbre. La source du motif est le nœud déterminant l'ensemble de sélection (ie. nom de collection) à laquelle est reliée la racine.

Définition 3.4 : Motif d'arbre (TreePattern)

Un motif d'arbres est un ensemble de *Node* et de *NodeLink* associé à une variable de définition, une source et une racine.

Un motif d'arbre est l'ensemble des nœuds et des liens de nœuds qu'un document XML doit posséder pour être filtré par celui-ci. La variable de définition correspond à son nom unique qui est utilisé en référence dans la requête *XQuery*. La source correspond au premier nœud du motif, alors que la racine correspond au(x) nœud(s) auquel(s) est associé les *XPath* définissant le domaine de sélection.

Représentation graphique d'un motif d'arbre

La représentation graphique d'un *TreePattern* s'appuie sur la modélisation des liens entre les différents nœuds par l'intermédiaire des *NodeLink*s. Pour compléter le *TreePattern*, nous encapsulons l'ensemble de nœuds dans une bulle à laquelle nous associons la variable de définition. Différents types de bulles sont proposés par la suite, ils dépendent du type de motif d'arbre à représenter.

Ainsi dans l'exemple de la figure 3.5, nous pouvons remarquer un motif d'arbre dont la source est le nœud *catalog*, et *book* en est la racine. Le lien entre *author* et *book* est

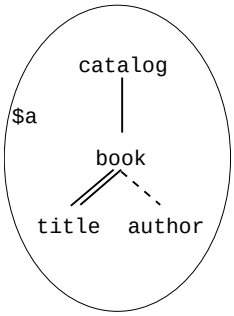


FIG. 3.5 – Motif d'arbre simple (TreePattern)

optionnel car il est représenté en pointillé et de type père/fils car le trait est simple. Le lien entre les nœuds *book* et *title* est de type ancêtre/descendant car il est double et obligatoire car il est en trait plein. Pour finir, nous pouvons voir que la variable de définition de ce motif d'arbre est *$a* annoté dans la bulle de ce *TreePattern*.

Différents types de motifs d'arbres

Comme nous pouvons le constater dans le langage de requêtes *XQuery*, plusieurs types d'expressions permettent de définir un ensemble d'éléments, chacun correspond à un motif d'arbres. Nous pouvons identifier deux types de clauses *for* (document cible ou redéfinition de variable) pour sélectionner des arbres et sous arbres dans un document XML, une clause *let* pour créer des agrégats sur des arbres, et la clause *return* pour créer un motif de création de document XML à partir des sélections. Nous pouvons donc définir quatre types de *TreePattern* :

- **Motif d'arbre source (SourceTreePattern ou STP)** : Un motif d'arbre source est un motif d'arbre définissant un document ou un ensemble de document XML cible. Il est associé à une racine.
  un motif d'arbre source correspond à un document XML ciblé par une clause for auquel est associé un XPath racine, permettant de sélectionner un sous-ensemble du document XML ciblé. Ce motif nous permet de définir un domaine qui peut être utilisé dans le reste du modèle.
- **Motif d'arbre Intermédiaire (IntermediateTreePattern ou ITP)** : Un motif d'arbre intermédiaire est un motif d'arbre définissant un sous-ensemble de nœuds à partir d'un motif d'arbre existant.
  Un motif d'arbre intermédiaire correspond à une définition d'une variable dans la clause for à partir d'une variable déjà déclarée. Cette déclaration permet de spécialiser un élément sur un nouveau motif d'arbres.
- **Motif d'arbre résultat (ReturnTreePattern ou RTP)** : Un motif d'arbre résultat est un motif d'arbre qui définit la construction d'un document XML. Chaque nœud du motif coïncide avec une balise du document XML résultat ou à un nœud texte.
  Un motif d'arbre résultat correspond à la clause return d'une requête XQuery permettant de produire un document XML. Nous pouvons identifier les balises de

ce document par les nœuds du motif, les attributs étant précédés par le symbole «@» et les éléments textuels par des guillemets. Les XPaths projetés sont associés à des nœuds vides de ce motif par des hyperliens directionnels (section 3.3.3)

– **Motif d'arbre d'agrégation (AggregateTreePattern ou ATP)** : Un motif d'arbre d'agrégation est un motif d'arbre générant un ensemble résultat temporaire. La fonction d'agrégat est associée au motif pour un traitement sur un ensemble d'arbres. Le motif d'arbre d'agrégation hérite des propriétés du *ReturnTreePattern*
Un motif d'arbre d'agrégation correspond à la clause let qui définit un traitement sur un ensemble d'arbres. Les fonctions d'agrégat sont appliquées sur un ensemble d'arbres produit par le motif d'arbre. Grâce aux règles de canonisation, les requêtes imbriquées sont définies dans des clauses let et produisent donc un ensemble d'arbres temporaires.

Motif d'arbre source et Motif d'arbre résultat

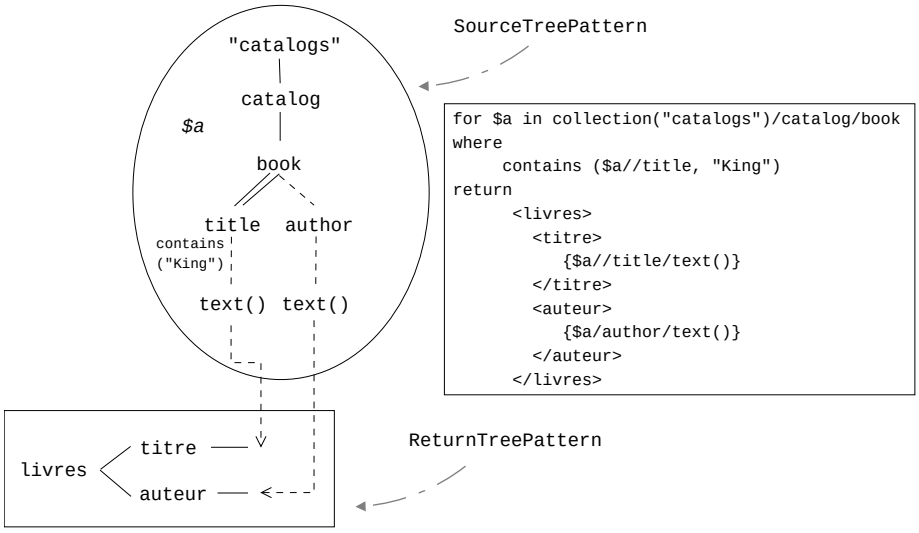


FIG. 3.6 – Motif d'arbre source (STP) et résultat (RTP)

Dans la figure 3.6, le motif d'arbre du haut représente la déclaration d'un motif d'arbre source associé à la variable *$a*. La source est la collection «*catalogs*» sur laquelle nous pouvons définir la racine */catalog/book*. Nous pouvons y voir le *Node title* avec la fonction *contains*, et le nœd *author* auquel est associé un *ProjectionHyperlink* vers le motif d'arbre résultat (en bas). Il est représenté à l'horizontale pour faciliter la lisibilité des hyperliens verticaux. De plus, cet arbre est inscrit dans un rectangle pour représenter la construction du résultat final. Ce motif d'arbre résultat contient une racine «*livres*», à laquelle ont été ajoutés deux *XPaths* : «*/titre*» et «*/auteur*». Nous avons associé à ces deux *nœuds* du *RTP* des *ProjectionHyperlinks* provenant des *nœuds* «*text()*» du *STP*. Ces deux *nœuds* sélectionnent en *XQuery* la valeur du texte contenue dans le nœud courant (les *nœuds title* et *author*).

Motif d'arbre intermédiaire et Motif d'arbre d'agrégation

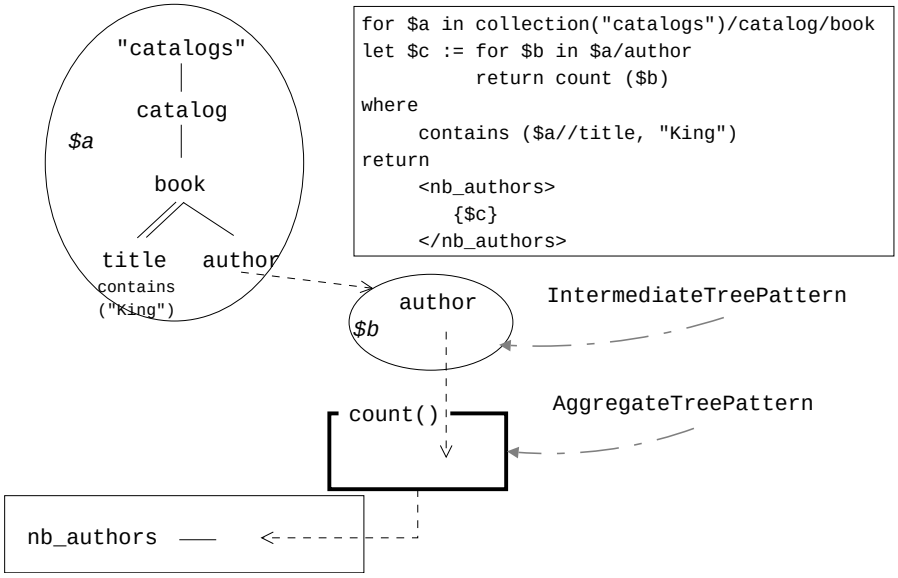


FIG. 3.7 – Motif d'arbre intermédiaire (ITP) et d'agrégat (ATP)

Dans l'exemple de la figure 3.7, nous pouvons observer quatre motifs d'arbre, un motif d'arbre source (en haut), un motif d'arbre résultat (en bas), ainsi qu'un motif d'arbre intemédiaire (milieu haut) et un motif d'arbre d'agrégation .
Le motif d'arbre intemédiaire de variable $b permet l'exploration du nœud *author* à partir de l'ensemble défini par le motif d'arbre source «$a». Le label de la racine est égal à celui du nœud *author*. Par ailleurs, nous pouvons remarquer que le *ExplorationHyperlink* reliant *author* à l'*ITP* est optionnel puisque celui-ci est en traits pointillés. Chaque auteur est projeté dans le motif d'arbre d'agrégation.
Le motif d'arbre d'agrégation prend les auteurs définis pour chaque livre du catalogue, et compte le nom d'auteurs (fonction «*count*»). L'hyperlien en dessous de ce motif d'arbre projette le résultat dans le motif d'arbre résultat.

Les motifs d'arbres permettent de définir des motifs pour les documents XML qui peuvent être utilisé dans notre modèle. Chacun des types de motifs permet de spécialiser le modèle en fonction des spécificités du langage de requêtes *XQuery*. Toutefois, il nous faut introduire les contraintes pour exprimer les filtres permettant de restreindre les documents XML.

3.3.2 Contraintes

Dans les requêtes *XQuery*, les contraintes expriment un filtre appliqué à un ou plusieurs nœuds d'un document XML. Elles permettent de réduire l'ensemble des arbres XML. De plus, les contraintes peuvent être appliquées aux variables dans les clauses *let*, entre deux chemins pour exprimer une contrainte de jointure entre deux

documents XML, ou même sur une autre contrainte (composition de fonctions). Nous introduisons donc le type *Constraint* :

Définition 3.5 : Contrainte (Constraint)

> Une contrainte est une restriction des solutions d'un ensemble d'arbre. Il peut être appliqué à un *Node*, un *TreePattern*, un *Hyperlink* ou une autre *Constraint*. Chaque contrainte est définie en tant que *Prédicat* ou *Fonction* reliée à l'élément concerné.

Une contrainte est associée à un nœud lorsque celle-ci est déclarée à l'aide d'un *XPath* unique. Lorsque deux *XPath*s sont présents un hyperlien de jointure est généré (section 3.3.3). Une composition de fonction génère deux contraintes. Les fonctions d'agrégat sont reliées aux motifs d'arbre d'agrégation. Par ailleurs, les liens entre les contraintes d'une clause *where* sont préservés par une *hyperlien de contraintes* comme nous pourrons le voir par la suite.

- **Fonction (Function)** : Une fonction est une contrainte avec un nom et un ensemble de liens vers des éléments. Ces éléments sont de type *Node*, *Constraint* ou *Constant*.
  Le type des fonctions permet de définir le nom de celles-ci, ainsi que leurs paramètres qui sont des liens vers des éléments. Ces éléments sont des nœuds pour les XPath, des compositions de fonctions ou des constantes.
- **Prédicat (Predicate)** : Un prédicat est un cas particulier de fonctions dont le résultat est booléen. Il prend deux éléments en paramètre et le nom de la fonction est déterminé par le comparateur. Ces éléments sont de type *Node*, *Constraint* ou une *constante*.
  Un prédicat permet de définir une relation de comparaison entre deux éléments. Nous pouvons donc comparer des valeurs de nœuds, de constantes et des résultats de contraintes. L'opération de comparaison est définie par les symboles suivant : $=, <, <=, >, >=, !=, >>, <<, is$.

La composition booléenne des contraintes d'une clause *where* est définie par l'association des *hyperliens de contraintes* (cf. section suivante). Une structure arborescente des contraintes permet de garder l'association booléenne avec les opérateurs AND et OR de la clause *where*.

Représentation Graphique

Notre modèle de représentation graphique intègre les contraintes sur les motifs d'arbres. Chaque contrainte est représentée en dessous du *nœud* auquel il est associé (les cas particuliers sont étudiés plus loin).

Les figures 3.6 et 3.7 illustrent la représentation d'une contrainte associée à un nœud. Les motifs d'arbre sources doivent contenir un titre avec le mot «*King*».

Nous avons donc défini les contraintes qui peuvent être représentées dans notre

modèle pour nous permettre de représenter les restrictions de résultat. Nous devons maintenant définir les liens permettant d'interagir entre tous les types que nous avons défini jusqu'ici : les *Hyperliens*.

3.3.3 Hyperliens

Les requêtes *XQuery* peuvent associer plusieurs variables par l'intermédiaire de contraintes de jointures, relier une clause *for* ou *let* à une clause *return*, définir de nouvelles variables, définir des expressions particulières tel que les opérations ensemblistes ou conditionnelles. Ainsi, nous introduisons les *hyperliens* qui nous permettront de représenter les liens entre les différents éléments composant notre modèle. Plusieurs types d'hyperliens seront introduits pour correspondre aux différentes spécificités du langage.

Définition 3.6 : Hyperlien (Hyperlink)

> Un hyperlien est un lien entre plusieurs éléments d'un *Tree Graph View*. Il représente l'association avec les *AssociationHyperlink* ou une transformation avec les *DirectionalHyperlink*

Un hyperlien relie deux (ou plus) éléments d'un *TGV* pour exprimer une association (restriction d'ensembles d'éléments) ou une transformation (d'un ensemble vers un autre).

Hyperliens Associatifs

Définition 3.7 : Hyperlien Associatif (AssociationHyperlink)

> Un hyperlien associatif est un hyperlien connectant deux éléments de même type pour représenter une association spécifique. Cette association filtre les résultats reliés.

Nous pouvons distinguer deux types d'hyperliens associatifs : Les hyperliens de jointure reliant deux nœuds, et les hyperliens de contraintes reliant deux contraintes.

- **Hyperlien de Jointure (JoinHyperlink)** : Un hyperlien de jointure est un hyperlien associatif reliant deux *Node* à l'aide d'une *Constraint*.
  Les hyperliens de jointures correspondent aux contraintes de jointures déclarées dans la clause where, ils permettent de joindre deux ensembles d'arbre à l'aide d'une contrainte de jointure. Les deux XPath utilisés génèrent deux nœuds qui seront reliés par l'hyperlien auquel est associé la contrainte. Les arbres XML sont filtrés par cet hyperlien s'ils vérifient la contrainte reliant les deux éléments.
- **Hyperlien de contrainte (ConstraintHyperlink)** : Un hyperlien de contrainte est un hyperlien associatif reliant sous forme arborescente, les contraintes et les hyperliens de contraintes à l'aide d'une opération booléenne. Chaque ensemble d'hyperlien de contrainte est associé à un *ReturnTreePattern* (ReturnTreePat-

tern).
Toutes les contraintes définies dans la clause where sont reliées ensembles par une séquence et/ou une hiérarchie d'opérations booléennes (and/or) formant un prédicat. Ce prédicat donne les contraintes qui doivent être vérifiée sur les ensembles d'arbres pour produire un résultat. Chaque clause where est associée à une clause return unique, représentant le niveau d'imbrication de la requête. Ainsi, les hyperliens de contraintes sont reliés au motif d'arbres résultat correspondant à la clause return. Cet hyperlien permet de connaître le niveau d'application d'une contrainte (qui peut être différent du niveau de déclaration de la variable associée), il permet aussi de définir les opérations booléennes reliant deux contraintes, ou hyperlien de contrainte afin de connaître la hiérarchie des contraintes.

Représentation Graphique

Un hyperlien de jointure est représenté par un lien, sous forme d'un trait plein (la contrainte est toujours obligatoire, contrairement aux éléments sélectionnés), entre deux éléments du modèle auquel est associée la contrainte. Un hyperlien de contraintes est représenté par une ligne courbe entre deux contraintes ou hyperliens de contraintes (au centre de l'hyperlien ciblé), celui-ci est en pointillé s'il est optionnel (opération booléenne *OR*), plein sinon (*AND*).

Dans un souci de visibilité du modèle, nous ne représentons les hyperliens de contraintes que dans le cas où une opération booléenne *OR* est présente (par défaut tous les hyperliens de contraintes seront obligatoires, et l'algèbre booléenne est commutative dans le cas des opérateurs *and*). Le lien reliant les hyperliens de contraintes au motif d'arbre résultat associé ne est représenté que dans le cas où une contrainte est liée à un niveau d'imbrication différent de celui de la déclaration de sa variable associée naturel (par défaut, les contraintes sont associées au premier RTP associé).

Dans la figure 3.8, deux motifs d'arbre source et un motif d'arbre résultat sont représentés. Les deux STP (SourceTreePattern) définissent les ensembles «*catalog*» ($a) et «*reviews*» ($b). Le titre du livre de $a est projeté vers le motif d'arbre résultat. Nous pouvons distinguer trois contraintes : deux fonctions *contains* («*King*» et «*Hobb*»), et un hyperlien de jointure entre les deux nœuds *author* de $a et $b. Ces trois contraintes sont reliées par des hyperliens de contraintes. Celui qui relie les deux fonctions *contains* est en pointillé montrant que pour le motif $a, soit le titre doit contenir le mot «*King*», soit le mot «*Hobb*». De plus, cet hyperlien est relié par un autre hyperlien de contrainte vers la contrainte de jointure («=»). Ainsi, si le motif d'arbre $a contient les mots requis, et si l'auteur est présent dans le motif $b, alors il est sélectionné et projeté dans un document dont la racine est «*books*».

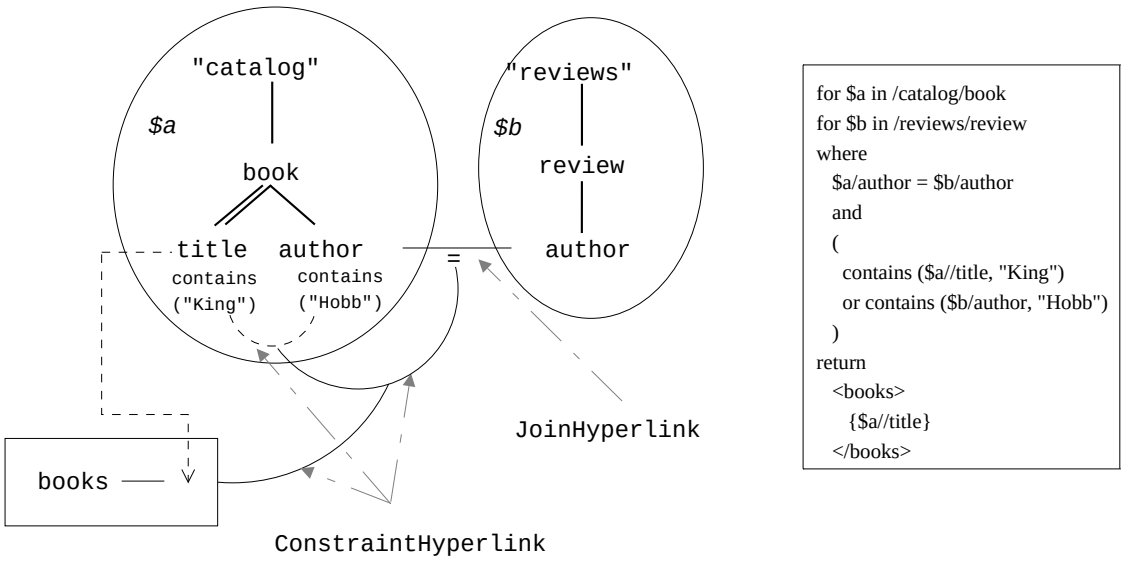


FIG. 3.8 – Hyperlien de contrainte et Hyperlien de jointure

Hyperliens Directionnels

Définition 3.8 : Hyperlien Directionnel (DirectionalHyperlink)

Un hyperlien directionnel est un hyperlien représentant une transformation injective entre plusieurs éléments. Il spécifie la transformation d'un ensemble d'éléments vers un élément unique.

Un hyperlien directionnel est une transformation injective car deux hyperliens directionnels ne peuvent avoir le même élément cible (l'élément ciblé par un hyperlien directionnel est unique, quelque soit le type d'hyperlien directionnel).

- **Hyperlien de projection (ProjectionHyperlink)** : Un hyperlien de projection est une hyperlien directionnel entre deux éléments de type *Node*. Il contient une information d'optionalité.
  Un hyperlien de projection permet de représenter la projection d'un nœud. Cela correspond à la déclaration d'un XPath dans la clause return, mais aussi dans certains cas dans la clause where. Le lien est optionnel dans le cas d'une clause return, obligatoire sinon.
- **Hyperlien d'exploration (ExplorationHyperlink)** : Un hyperlien d'exploration est un hyperlien directionnel entre un élément de type *Node* et un élément de type *IntermediateTreePattern* (IntermediateTreePattern). Il contient une information d'optionalité.
  Un hyperlien d'exploration représente le parcourt d'un nœud par un nouveau motif d'arbre. Il correspond à la déclaration d'une variable dans la clause for à partir d'une variable déjà définie. Cet hyperlien relie donc un nœud à un motif d'arbre intermédiaire, il définit l'exploration du nœud par le nouveau motif d'arbre. L'hy-

perlien est optionnel lorsque la clause for est imbriquée par rapport à la variable explorée.

- **Hyperlien de Généralisation (GeneralizedHyperlink)** : Un hyperlien de généralisation est un hyperlien directionnel entre un élément de type *AggregateTreePattern* (AggregateTreePattern) et un élément de type *Node*. Il contient une information d'optionalité.
  Un hyperlien de généralisation permet de regrouper l'ensemble des résultats contenu dans un motif d'arbre d'agrégation pour retourner le résultat agrégé. Ainsi, nous pouvons retourner les imbrications de requêtes et les valeurs retournées par les fonctions. Cet hyperlien peut être obligatoire lorsque la fonction d'agrégat est déclarée dans la clause where.
- **Hyperlien Ensembliste (SetHyperlink)** : Un hyperlien ensembliste est un hyperlien directionnel entre un ensemble de types *TreePattern* vers un élément de type *Node*, relié à l'aide d'une contrainte ensembliste. Cette contrainte peut prendre les valeurs suivantes : *Union*, *Intersection* ou *Différence*.
  *Cet hyperlien représente une opération ensembliste entre plusieurs domaines définis par des motifs d'arbres pour les projeter sur un nœud unique qui prend pour valeur l'ensemble résultant de cette opération. XQuery définit l'opération ensembliste à l'aide de parenthèses et des opérateurs «**U**(nion)», «**I**(ntersect)» et «**E**(xcept)».*
- **Hyperlien Conditionnel (IfThenElseHyperlink)** : Un hyperlien conditionnel est un hyperlien directionnel entre deux éléments, de type *Node* ou *AggregateTreePattern*, une *Constraint* et un *Node* de projection. La contrainte peut être de type *Constraint* ou *ConstraintHyperlink*.
  Cet hyperlien permet de représenter une opération conditionnelle présente dans la clause return des requêtes XQuery, sous la forme de l'expression «If/Then/Else». Les deux éléments reliés à l'hyperlien sont de type Node lorsque la projection est un XPath, et de type AggregateTreePattern lorsque nous avons une requête imbriquée. La contrainte est une contrainte ou un hyperlien de contrainte dans le cas d'un ensemble de contraintes. Comme le langage XQuery est un langage opérationnel, il permet de donner le premier élément si la contrainte est vraie, le second si elle est fausse.

Représentations graphiques

La figure 3.9 nous montre quatre *TreePatterns* reliés par des hyperliens de types différents. Le premier *tp* ($a) s'applique sur la source «*catalog*» et la racine «*book*», nous appliquons à l'élément «*title*» la fonction «*contains("King")*», et que nous projetons sur le *tp* de reconstruction en bas. Cette projection est la représentation d'un hyperlien de projection. Le motif d'arbre en haut à droite est une exploration du premier, alors que le dernier motif d'arbre encadré est projeté dans le *tp* de reconstruction.

Cet hyperlien est représenté en pointillé pour exprimer l'optionalité de ce lien. En effet, une projection sur un résultat n'est pas obligatoire contrairement à la sélection (respectivement la clause *return* et la clause *for*).

De plus, la figure 3.9 illustre la représentation d'un hyperlien d'exploration. Le pre-

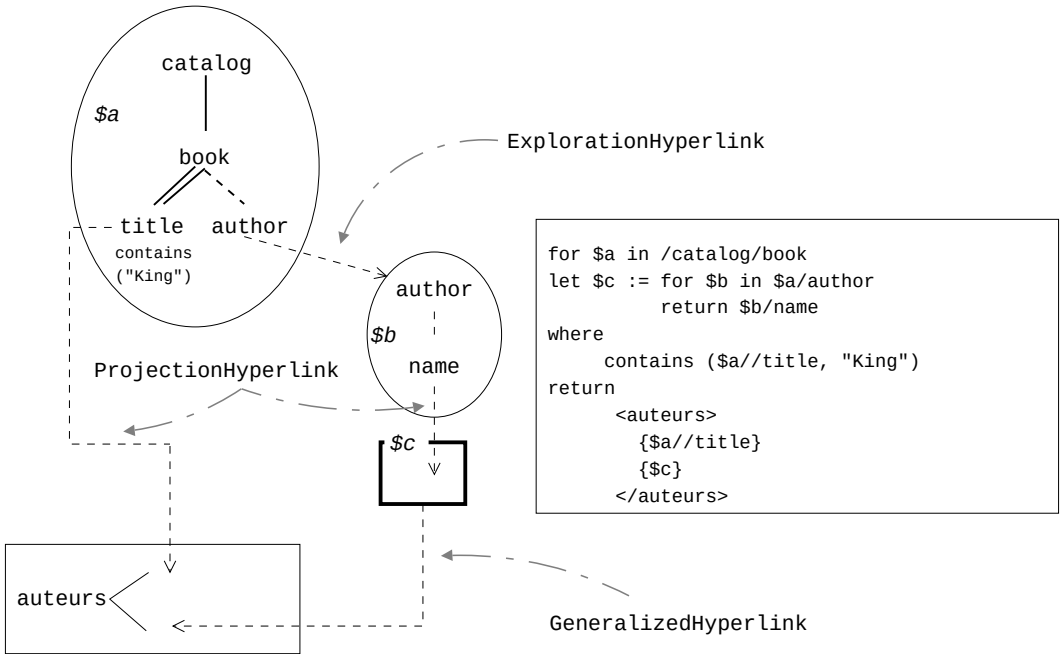


FIG. 3.9 – Hyperlien de projection

mier motif d'arbre ($a) est le *TreePattern* de définition de domaine, dont le nœud *author* est lié au second motif d'arbre ($b) par un hyperlien d'exploration. Ce dernier contient les nœuds *author* et *name* avec la nouvelle variable $b, le lien d'exploration spécifie la définition d'un nouveau domaine autour de l'élément *author* auquel il est lié. Ensuite, un hyperlien relie le nom à un motif d'arbre d'agrégation qui est lui-même projeté dans le motif d'arbre résultat.

Ainsi, nous obtenons un hyperlien d'exploration reliant un nœud à un motif d'arbre, nous permettant d'explorer le domaine de définition d'un élément dans le nouvel ensemble.

Enfin la figure 3.9 représente un hyperlien de généralisation. Ainsi, le motif d'arbre $a se projette dans la clause de reconstruction de résultat, et le *TreePattern* $b se projette sur le quatrième motif d'arbre $c. Ce dernier est relié au motif d'arbre de reconstruction de résultat grâce à un hyperlien de généralisation pour représenter l'imbrication du résultat des noms d'auteur.

L'hyperlien de généralisation provient de la projection de la clause *let* dans la clause *return* (nous étudions le cas du traitement de la clause *let* dans la section 3.4.4.4 sur les *AggregateTreePattern*). Ainsi, pour cette requête, tous les noms d'auteurs seront imbriqués dans la balise *auteurs* grâce à la clause *let*. Cela est représenté dans la figure 3.9 par l'hyperlien de généralisation entre le cadre en gras et la clause de reconstruction de résultat.

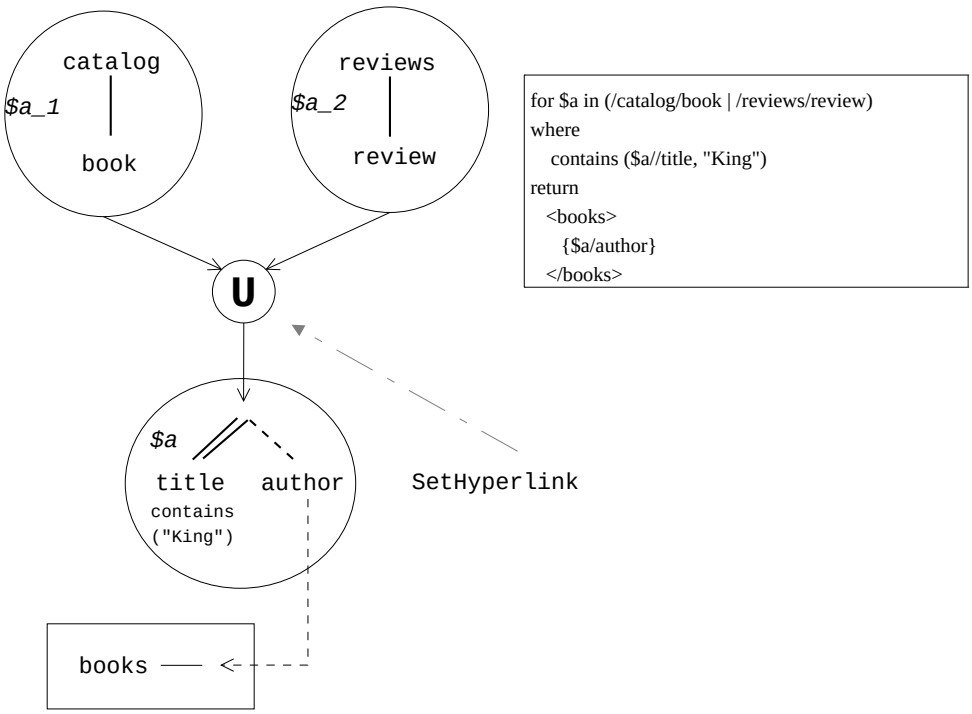


FIG. 3.10 – Hyperlien Ensembliste

Nous pouvons voir dans la figure 3.10 une composition particulière de *TreePattern*s sous forme d'une union des deux motifs d'arbre supérieurs vers le *TreePattern* central. les *TreePattern*s $a_1 et $a_2 définissent un domaine qui est uni par l'opérateur d'union situé au centre. Le résultat est projeté dans le *TreePattern* $a sur lequel nous effectuons la fonction «*contains ($a/title, "King")*» et la projection du *Node author* vers le *TreePattern* de reconstruction (section 3.4.4.2).

Nous pouvons observer l'union (U) dans la clause *for* sur les deux domaines */catalog/book* et */reviews/review*, permettant de définir le*TreePattern* $a. L'intersection est représentée à l'aide du symbole I, ainsi que l'exception par E.

Comme nous pouvons le voir dans la figure 3.11, nous avons une représentation particulière d'un motif d'arbre relié vers la reconstruction de résultat. L'opérateur central représenté sous forme d'un losange est l'hyperlien conditionnel (*IfThenElseHyperlink*), nous pouvons apercevoir les trois liens nécessaires pour évaluer la condition.

- Le lien annoté d'un *If* reliant l'opération de comparaison à l'hyperlien de condition. Cela nous permet d'identifier l'opération de comparaison ;
- Le lien de généralisation de gauche annoté d'un *Then* permet de récupérer le nœud *author* projeté dans le *TreePattern* en gras[2]. Ce résultat est retourné si la condition «*contains ($a//title, "King")*» est vraie ;
- Le lien de généralisation de droite *Else* permet de retourner un *TreePattern* conte-

[2]cf section 3.4.4.4 pour les*AggregateTreePattern*

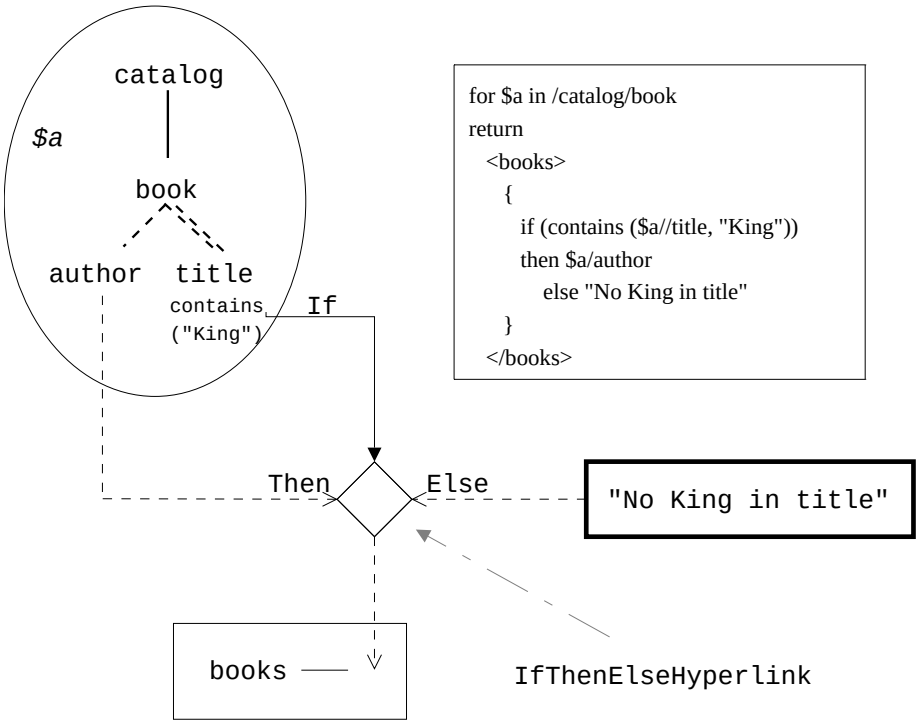


FIG. 3.11 – Hyperlien Conditionnel

nant la chaîne de caractère «*No King in title*». Ce résultat est retourné lorsque la condition «*contains ($a//title, "King")*» est fausse.

Ainsi, nous pouvons représenter les *IfThenElseHyperlink*s dans nos *TGV*s à l'aide des différents hyperliens permettant de relier nos différents éléments.

Nous avons donc déterminé les hyperliens qui permettent de relier les différents éléments composant notre modèle. Nous pouvons distinguer les hyperliens correspondant à une association (*AssociationHyperlink*) correspondant au traitement sur les arbres pour sélectionner les résultats (jointure et contraintes). Les hyperliens directionnels (*DirectionalHyperlink*) produisent des arbres résultats qui pourront être utilisés dans le modèle (projection, ensembles, agrégation, conditionnel, exploration).

La figure 3.12 représente la classification de tous les liens possibles dans notre modèle. Nous pouvons classer trois grandes familles : *NodeLink*, *ConstraintLink* et *Hyperlink*. Les *NodeLink*s sont les liens entre les nœuds, les *ConstraintLink* sont les liens entre une contrainte et un nœud, et les hyperliens sont des liens entre les éléments d'un *TGV*. Parmi les hyperliens, nous pouvons distinguer les deux classes principales, les hyperliens directionnels et les hyperliens associatifs.

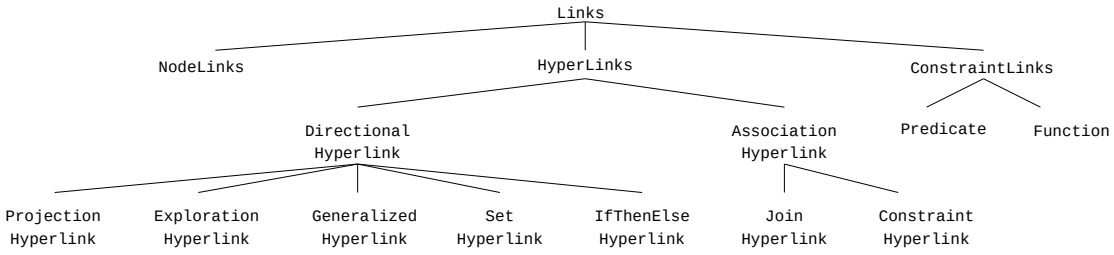


FIG. 3.12 – Hiérarchie des liens

3.3.4 Tree Graph Views

Un *Tree Graph View* (TGV) est une représentation de requêtes *XQuery*. Il est composé de tous les éléments que nous avons vu précédemment, c'est à dire : de motifs d'arbres, de contraintes et d'hyperliens. Les arbres XML traités par le *TGV* seront filtrés par les motifs d'arbre source (*SourceTreePattern*), restreints grâce à l'ensemble des contraintes associées (*ConstraintHyperlink*) aux motifs d'arbre résultat (*ReturnTreePattern*). Le résultat final est produit par un *RTP* pour donner un nouvel ensemble d'arbre XML.

Définition 3.9 : TreeGraphView (TGV)

Un *Tree Graph View* est un ensemble de motifs d'arbres liés entre eux par des hyperliens, et restreints par des contraintes. Il contient un ensemble de motifs d'arbre source en entrée, et un motif d'arbre résultat en sortie.

Ainsi, nous pouvons modéliser une requête *XQuery* canonisée à l'aide des *TGV* en transformant chaque clause par un type qui lui est associé. Ces types génèrent des hyperliens permettant de produire un graphe. Ce graphe est une vue de la requête qui pourra ensuite est transformé en algèbre pour produire être évaluée et ainsi produire des documents XML.

3.3.5 Conclusion sur le modèle

Nous avons donc introduit une représentation des requêtes *XQuery* que nous avons nommé *TGV*. Les *TGV*s sont un ensemble de *TreePattern*s qui correspondent à des filtres applicables à un document XML. Chaque *TreePattern* est composé de nœuds (équivalent d'une balise en XML) associés entre eux par des *NodeLink*s qui représentent les liens entre les différents nœuds de ce *TreePattern*. Des contraintes peuvent être ajoutées aux nœuds pour restreindre l'ensemble de définition d'un *TreePattern*. Les *TGV*s peuvent contenir plusieurs types de *TreePattern* correspondant

aux différents types requis dans une requête *XQuery*, l'ensemble de ces *TreePatterns* sont inter-connectés par l'intermédiaire d'hyperliens représentant les relations définies dans le langage de requêtes *XQuery*. Nous allons maintenant proposer une formalisation de notre modèle à l'aide des Types Abstraits de Données (ADT), nous permettant de définir de façon rigoureuse notre modèle.

3.4 Formalisation des Tree Graph Views

Un *Tree Graph View* (TGV) est une représentation logique des requêtes XQuery. Il permet de déterminer chacun des éléments nécessaires à la formulation d'une requête. Pour cela, il nous faut définir un ensemble de définition pour ces différents éléments sous forme de Types Abstraits de Données (*Abstract Data Type - ADT*).

Un type abstrait de données (ADT) [Guttag 1975] est une spécification mathématique d'un ensemble de données et de l'ensemble des opérations qui peuvent y être effectuées. Un tel type de données est qualifié d'abstrait car il correspond en fait à un cahier des charges qu'une structure de données doit ensuite implémenter. Il est défini par un triplet Opérations/Axiomes/Pré-conditions.

Les *opérations* déterminent les actions qui sont possibles sur ce Type Abstrait de Données. Une opération est identifiée par son nom, les informations en entrée et la valeur de retour. Chaque opération est caractérisée par un ou plusieurs *axiomes* qu'à tout instant les valeurs du Type Abstrait doivent vérifier (avant et après l'opération). Les axiomes sont partagés entre les constructeurs et les opérateurs. Un constructeur défini les valeurs du type abstrait. Chaque axiome montre l'application d'une opération sur un constructeur ou une opération. L'instance d'un type abstrait est valide si tous ses axiomes sont validés. Les *pré-conditions* définissent les domaines de validité des opérations. En supposant qu'à l'entrée d'une opération ses pré-conditions soient vraies, alors l'implémentation garantit qu'à la sortie les axiomes seront toujours satisfaits.

Par la suite nous considérons les types prédéfinis *string* (chaîne de caractères), *boolean* (booléen vrai ou faux) et *int* (entier).

3.4.1 Motifs d'arbre

Comme nous avons pu le voir dans la section 3.3.1, un motif d'arbre est composé de nœuds et de liens de nœuds. Chaque nœud représente un élément d'un *XPath*, les liens de nœuds représentent le lien d'ascendance entre deux éléments. Nous définissons les Types Abstraits de Données pour chacun de ces types *Node* (TA 1) pour les nœuds, *NodeLink* (TA 5) pour les liens de nœuds et les *TreePattern* (TA 6) pour les motifs d'arbres. Nous avons besoin de définir le type abstrait pour les *XPath* (TA 5) afin de nous permettre de les insérer dans un *TreePattern*.

3.4.1.1 Nœuds

Un nœud représente une information sur une balise que doit contenir un document XML, il correspond à un élément contenu dans un *XPath*. Pour représenter ce nœud, il nous faut garder certaines informations nous permettant de former un *TreePattern* (un ensemble de nœuds et de *NodeLink*). Afin de définir ce nœud, nous devons définir le *Type Abstrait Node* qui nécessite une chaîne de caractère correspondant au nom de la balise.

TA 1 Node N
Utilise : String
Opérations
O1 createNode String → N
O2 label N → string
Axiomes : $s \in string$;
A1 label (createNode (s)) = s

L'opération *O1* nous permet de définir un nœud à l'aide de son label. Ce dernier peut être récupéré grâce à l'opération *O2* appliquée à un nœud et son axiome *A1*.

Ainsi, ce type abstrait *Node* nous permet de créer un nœud à partir d'un label. Nous avons maintenant besoin de gérer les relations entre les différents nœuds.

3.4.1.2 Axe

Comme nous avons pu le voir dans la définition des associations entre nœuds, plusieurs axes sont possibles. Ce type abstrait de données permet de garder le nom de l'axe qui est donné dans le chemin *XPath*. La chaîne de caractère utilisée lors de la construction du TA Axis doit être compris dans l'ensemble de valeur {*'ancestor', 'ancestor-or-self', 'child', 'descendant', 'descendant-or-self', 'following', 'following-sibling', 'preceding', 'preceding-sibling'*}, correspondant aux neufs axes possibles parmi les liens de nœuds (les quatres autres axes de *XPath* sont considérés comme des nœuds). L'opération de création de l'axe *O1* prend la chaîne de caractère de l'axe, qui peut être récupérée grâce à l'opération *O2* déterminée par l'axiome *A1*.

TA 2 Axis A
Utilise : String $\in$ {'ancestor', 'ancestor-or-self', 'child', 'descendant', 'descendant-or-self', 'following', 'following-sibling', 'preceding', 'preceding-sibling'}
Opérations
O1 createAxis String → A
O2 getAxis A → String
Axiomes : $String \in s$;
A1 getAxis (createAxis (s)) = s

3.4.1.3 Liens de nœuds

Un lien de nœuds (*NodeLink*) donne le lien d'ascendance entre deux éléments de type *Node*. Il doit contenir les informations de parenté (parent/enfant ou ancêtre/descendant), l'optionalité du nœud par rapport à son père, et son ordre de déclaration dans la requête *XQuery*[3].
L'ascendance d'un nœud dépend du chemin qui a été introduit pour la construction de ce *NodeLink*. Si celui-ci est déterminé par un '*/label*' le lien sera de type parent/enfant, alors que s'il est déterminé par '*//label*' il sera de type ancêtre/descendant.
L'optionalité d'un *NodeLink* défini si un nœud est déterminant ou non dans l'apparition d'un résultat obtenu par l'application de l'ensemble des nœuds. Nous retrouvons cette notion d'optionalité dans la clause *return* d'une requête *XQuery*.

TA 3 NodeLink NL

Utilise : N, Axis, int, boolean

Opérations

O1	createNodeLink	N × N × int × Axis × boolean	→ NL
O2	getPosition	NL	→ int
O3	getLinkAssociation	NL	→ Axis
O4	isMandatory	NL	→ boolean
O5	getPreviousNode	NL	→ N
O6	getNextNode	NL	→ N
	`fonctions ajoutées au TA Node`		
O7	isRoot	N	→ boolean
O8	getNodeLink	N	→ NL
O9	getNextNodeLink	N × int	→ NL
O10	getSizeNodeLink	N	→ int

Pré-conditions : $n \epsilon N$; $i \epsilon int$;

P1 define (getNodeLink (n)) $\Longleftrightarrow \neg$ isRoot (n)

P2 define (getNextNodeLink (n, i)) $\Longleftrightarrow$ i >= 0 && i < getSizeNodeLink (n)

Axiomes : $s \epsilon string$; $n_1, n_2 \epsilon N$; $pos \epsilon int$; $a \epsilon Axis$; $m \epsilon boolean$; , $nl \epsilon NL$;

A1 getPosition (createNodeLink (n_1, n_2, pos, a, m)) = pos

A2 getLinkAssociation (createNodeLink (n_1, n_2, pos, a, m)) = a

A3 isMandatory (createNodeLink (n_1, n_2, pos, a, m)) = m

A4 getPreviousNode (createNodeLink (n_1, n_2, pos, a, m)) = n_1

A5 getNextNode (createNodeLink (n_1, n_2, pos, a, m)) = n_2

`axiomes ajoutés aux TA Node`

A6 isRoot (getNextNode (nl)) = false

A7 isRoot (createNode (s)) = true

A8 getNodeLink (getNextNode (nl)) = nl

A9 getNextNodeLink (getPreviousNode (createNodeLink (n_1, n_2, pos, a, m)) , pos) = createNodeLink (n_1, n_2, pos, a, m)

A10 getSizeNodeLink (createNode(s)) = 0

A11 getSizeNodeLink (getPreviousNode (createNodeLink (n_1, n_2, pos, a, m))) = getSizeNodeLink (n_1) + 1

Nous définissons maintenant le Type Abstrait de Données 5 dans lequel, l'opération *O1* permet de créer un *NodeLink* qui sera un lien entre deux nœuds. Nous devons

[3] l'ordre de déclaration est défini dans la requête *XQuery* par l'ordre de déclaration des *XPath*. Cette information n'a d'influence que sur l'ordre de recherche des éléments.

donner la position de ce lien par rapport au nœud père, l'ascendance de ce lien (parent/enfant ou ancêtre/descendant), et son optionalité (obligatoire ou optionnelle).

Les opérations *O2*, *O3* et *O4* nous permettent de récupérer les trois dernières informations spécifiées dans le constructeur du *Type Abstrait*. Les axiomes *A1*, *A2* et *A3* déterminent respectivement la position du *NodeLink* (avec l'entier), son ascendance (premier booléen) et son optionalité (deuxième booléen).
Grâce aux opérations *O5* et *O6* nous pouvons récupérer les nœuds père et fils. Leurs axiomes respectifs sont *A4* et *A5* pour spécifier leur ordre de création.

Les quatre opérations *O7* à *O10* peuvent être ajoutées au *Type Abstrait Node* pour enrichir notre modèle.
- L'opération *O7* détermine si un nœud est racine. Dans ce cas, il n'a pas de *NodeLink* pour le relier à un nœud père. L'axiome *A6* nous permet de dire que si un nœud a un *NodeLink* fils, alors il n'est pas racine. L'axiome *A7* nous permet de dire qu'un nœud créé est toujours racine. Ainsi, lorsque l'on créé un *NodeLink* entre n_1 et n_2, alors n_2 n'est plus racine ;
- L'opération *O8* nous permet de récupérer le *NodeLink* d'un nœud. Grâce à l'axiome *A8* nous pouvons en déduire que ce *NodeLink* est le lien qui le relie au nœud père. En effet, si pour un nœud n, nous récupérons son *NodeLink* (*O8*), alors le nœud fils de ce *NodeLink* est n. La pré-condition *P1* permet de définir que l'opération *O8* n'est applicable à un nœud que s'il n'est pas racine ;
- Pour récupérer le n^{ème} *NodeLink* fils d'un nœud, nous ajoutons l'opération *O9*. Il est déterminé par l'axiome *A9* qui nous montre que le *NodeLink* fils associé à un nœud père à la position *pos* est le *NodeLink* que l'on peut ajouter à n_1 à la position *pos*. La pré-condition *P2* définit que l'opération *O9* n'est appliquable que dans le cas où l'entier i est compris entre 0 et le nombre de *NodeLink* lié à ce nœud, donc que ce *NodeLink* a bien été ajouté ;
- L'opération *O10* nous donne le nombre de *NodeLink* fils qui sont associés à un nœud. L'axiome *A10* initialise ce nombre à zéro lors de la création de ce nœud, alors que l'axiome *A11* augmente ce nombre lorsqu'un nouveau *NodeLink* est associé à ce même nœud.

Ainsi, pour un *NodeLink nl* que nous définissons par *createNodeLink* (n_1, n_2, *pos, c, m)*, nous pouvons connaître la position *pos* du nœud n_2 par rapport à son père n_2 grâce à la fonction *getPosition(nl)*. La relation entre n_1 et n_2 est de type parent/enfant si la fonction *isDirectSon (nl)* est vrai, sinon celle-ci est de type ancêtre/descendant. La relation entre n_1 et n_2 est obligatoire si la fonction *isMandatory (nl)* est vraie.

3.4.1.4 XPath

Afin d'insérer un ensemble d'éléments dans un même domaine, nous avons besoin de définir un *XPath*, qui permet de connaître la hiérarchie de ses éléments. Chacun d'eux pourra être ajouté à un *TreePattern* et formera les nœuds et les liens de nœuds (cf TA 6). Nous considérons ici des *XPath canonisés* et *abrégés* représentant un chemin (c'est à dire uniquement composé de nom d'éléments). Ainsi, un *XPath*

est un ensemble d'*éléments* associés les uns aux autres par un axe de parcours. Cet *XPath* doit pouvoir nous permettre de gérer ces *éléments* sous forme de file.

TA 4 Element E

Utilise : string, Axis

Opérations

O1	createElement	string × Axis	→ E
O2	labelElement	E	→ string
O3	getAxis	E	→ Axis

Axiomes : $s \in string$; $a \in Axis$;

A1	labelElement (createElement (s, a))	= s
A2	isChild (createElement (s, a))	= a

L'opération *O1* nous permet de créer un élément dont le label est défini par l'opération *O2* et l'axiome *A1*. L'axe est défini par l'opération *O3* et l'axiome *A2*. Ainsi, nous obtenons des chemins composés d'éléments et d'axes.

TA 5 XPath XP

Utilise : E, boolean

Opération

O1	createXPath		→ XP
O2	first	XP	→ E
O3	last	XP	→ E
O4	addLast	XP × E	→ XP
O5	deleteFirst	XP	→ XP
O6	isEmpty	XP	→ boolean

Pré-conditions : $xp \in XP$;

P1	define (first (xp))	⟺ isEmpty (xp) = false
P2	define (last (xp))	⟺ isEmpty (xp) = false
P3	define (deleteFirst (xp))	⟺ isEmpty (xp) = false

Axiomes : $x \in XP$; $e \in E$;

A1	first (addLast (createXPath, e))	= e
A2	first (addLast (x, e))	= first (x)
A3	last(addLast (createXPath, e))	= e
A4	last(addLast (x, e))	= e
A5	deleteFirst (addLast (createXPath, e))	= createXPath
A6	isEmpty (createXPath)	= true
A7	isEmpty (addLast (x, e))	= false

L'opération *O1* nous permet de créer un *XPath*. Le premier élément peut être récupéré grâce à *O2*, celui-ci correspond au premier élément de cet *XPath* (*A1* et *A2*). Le dernier élément peut être récupéré à partir de l'opération *O3*, défini à partir des axiomes *A3* et *A4*. Un élément est ajouté à l'*XPath* en queue de file grâce à l'opération *O4* et l'axiome *A3* et *A4*. La suppression d'un élément se fait par la tête de la file, grâce à l'opération *O5* et l'axiome *A5*. Pour finir, l'opération *O6* nous détermine si un *XPath* est vide. Les axiomes *A6* et *A7* nous permettent de déduire qu'un *XPath* est vide à sa création et non vide lorsqu'on a ajouté un élément à celui-ci.

Ainsi, un *XPath* contient des *Element*s qui représentent les informations de *label* et d'*ascendance* entre l'élément courant et son prédécesseur. Le Type Abstrait *XPath* est traité comme un type file que l'on vide élément par élément. Nous pouvons donc, à partir du *Type Abstrait XPath*, introduire celui correspondant au *TreePattern*.

3.4.1.5 TreePattern

Un *TreePattern* est un ensemble de nœuds (*Node*) interconnectés par un ensemble de *NodeLink*s. Un *TreePattern* est identifié par une variable de définition (en XQuery celle-ci est déclarée dans la clause *for* ou *let*). Un *TreePattern* permet de créer chacun des nœuds qui le compose ainsi que les liens qui les unissent. Lorsque l'on considère un *XPath* (TA 5), chaque *Element* de celui-ci se traduira par à un nœud (TA 1) et le lien entre deux nœuds (TA 5) est donné par l'information *isChild*.

La racine de ce *TreePattern* est le nœud auquel seront ajoutés chacun des *XPath* associés au *TreePattern*. La source est définie par l'ensemble des nœuds associés à la variable de définition lors de sa création, il correspond à l'*XPath* de création de la variable en *XQuery*. Par exemple, pour la requête *for $i in /catalogs/catalog/book*, la source est */catalogs/catalog* et la racine *book*.

TA 6 TreePattern TP

Use : string, XP, N, Axis, int, boolean

Operations

O1	createTP	string × XP	→ TP
O2	getVariable	TP	→ string
O3	getRoot	TP	→ N
O4	getSource	TP	→ N
O5	getNode	TP × XP	→ N
O6	addChildNode	N × string × int × boolean × Axis	→ N
O7	addXPathToNode	N × XP × boolean	→ N
O8	addXPathToTreePattern	TP × XP × boolean	→ TP

Pré-conditions : $n \in N$; $xp \in XP$; $b \in boolean$;

P1 define (addXPathToNode (n, xp, b)) $\Longleftrightarrow$ isEmpty (xp) = false

Axiomes : $tp \in TP$; $n \in N$; $xp \in XP$; $s \in string$; $a \in Axis$; $m \in boolean$; $e \in E$;

A1 getVariable (createTP (s, xp)) = s

A2 label (getRoot (createTP (s, addLast (xp, e)))) = labelElement (e)

A3 isDirectSon (getNodeLink (getRoot (createTP (s, addLast (
createXPath, e))))) = isChild (e)

A4 label (getSource (createTP (s, xp))) = labelElement (first (xp))

A5 isDirectSon (getNodeLink (getSource (createTP (s, addLast (xp, e))),1))
= isChild (e)

A6 addChildNode (n, s, m, a) = getChildNode (createNodeLink (
n, createNode (s), getMaxNodeLink (n), m, a))

A7 addXPathToNode (n, xp, m) = addXPathToNode (
addChildNode (n, getLabel (first (xp)),
getAxis (first (xp)), m), deleteFirst (xp), m)

A8 getNode (addXPathToTreePattern (tp, xp, m), xp)
= addXPathToNode (getRoot (tp), xp, m)

A9 getLabel (getNode (addXPathToTreePattern (tp, xp), xp))
= getLabelElement (last (xp))

A10 isDirectSon (getNodeLink (getNode (addXPathToTreePattern (tp, xp), xp)))
= isChild (last (xp))

L'opération *O1* nous permet de voir qu'un *TreePattern* est défini par son nom de variable (*O2* et *A1*), ainsi que d'une source et d'une racine déterminées par un *XPath* (*O3* et *O4*). L'axiome *A2* définit la racine par le dernier élément de l'*XPath*

donné. La source est déterminée à partir du premier élément de cet *XPath* grâce à l'axiome *A4*. Le lien d'ascendance entre la racine et son prédécesseur est défini par l'*XPath* donné à la création (*A3*). De même, le lien d'ascendance entre la source et son fils est défini par l'axiome *A5*.

Un nœud d'un *TreePattern* peut être récupéré à l'aide d'un *XPath* en utilisant l'opération *O5* : *getNode*. L'axiome *A9* définit le label du dernier élément créé à l'aide du *XPath xp*, est le nœud qui est associé à ce même *XPath*.

Les opérations *O6*, *O7* et *O8* permettent d'ajouter un nœud dans un *TreePattern*. *O6* créé un nouveau nœud que l'on associe à un nœud de *tp*, qui peut-être identifié grâce à l'axiome *A6*. La place du *NodeLink* associé est le nombre de *NodeLink*s liés au nœud père. L'opération *O7* permet d'ajouter un *XPath* à un nœud, l'axiome *A7* nous montre que l'on construit récursivement le *TreePattern* à l'aide de chacun des éléments du *XPath* en l'ajoutant au nouveau nœud créé grâce à *O6*, la condition d'arrêt étant la pré-condition *P1* pour un *XPath* vide. L'opération *O8* associe un *XPath* à un *TreePattern*, permettant d'initialiser l'opération *O7* sur le nœud racine (*A8*).

Le *Type Abstrait* est complété par les axiomes *A9* et *A10* qui nous permettent de définir que le nœud associé à un *TreePattern* grâce à son *XPath* «*xp*» définissant son label (*A9*) et son axe (*A10*) (dernier élément de *xp*).

Ainsi, un *TreePattern* est un arbre contenant les informations sur l'ensemble des nœuds applicables à un même domaine. Cet arbre nous permet d'isoler toutes les informations nécessaires pour sélectionner les documents. Afin d'aider à la compréhension de la construction d'un *TreePattern* à l'aide des *XPath*, nous proposons quelques algorithmes disponibles en Annexe E.

Nous avons donc formalisé les types permettant de définir un motif d'arbre, composé de nœuds et de liens de nœuds produits à partir des *XPath*. Avant de définir les types abstraits de données pour les différentes classes de motifs d'arbre, nous devons définir les contraintes et les hyperliens qui sont utilisés par les *TreePattern*.

3.4.2 Contraintes

Comme nous avons pu le voir dans la définition des contraintes (section 3.3.2), ces dernières permettent de restreindre l'ensemble de données en fonction des valeurs ou des informations qui la compose. Ces contraintes peuvent être un prédicat ou une fonction. Nous devons donc définir un type abstrait général *Constraint* (TA 7) dont les prédicats et les fonctions pourront hériter. Ensuite, les liens de contraintes (TA 11 relieront une contrainte à un élément du tgv (nœud, contrainte, motif d'arbre).

Une contrainte est déclarée par son nom (*O2*), il est déterminé par l'axiome *A1*. A partir de ce type générique, nous pouvons alors spécifier par héritage les deux types principaux de contraintes : *Prédicats* et *Fonctions*.

TA 7 Constraint C

Utilise : string

Opérations

O1	createConstraint	string	→ C
O2	getConstraintName	C	→ string

Axiomes : $s \in string$;

A1 getConstraintName (createConstraint (s)) = s

3.4.2.1 Prédicats

Un prédicat est défini par un opérateur de comparaison (=, <, >, <=, >=, ! =, <<, >>, ∗, +, −, /), et une constante de restriction. Nous utilisons le type "chaînes de caractères" pour les représenter.

TA 8 Predicate P extends Constraint

Utilise : string

Opérations

O1	createPredicate	string × string	→ P
O2	getComparator	P	→ string
O3	getConstant	P	→ string

Axiomes : $comp \in \{'<','<=','=','>=','>','<<','>>','*','+','-','/'\}$; $cons \in string$;

A1 getComparator (createPredicate (comp, cons)) = comp

A2 getConstant (createPredicate (comp, cons)) = cons

Ainsi, nous pouvons voir qu'un prédicat est défini par son comparateur (*O2* et *A1*) et par une constante (*O3* et *A2*).

3.4.2.2 Fonctions

Le deuxième type de contrainte applicable à un nœud est la fonction. Pour représenter celle-ci, nous avons besoin du nom de cette fonction, et de l'ensemble des paramètres qui lui sont associés. Ces paramètres seront représentés par le type abstrait *ListParameter* qui est une liste de chaînes de caractères associées à une contrainte.

TA 9 ListParameter LP

Utilise : string, int

Operations

O1	createList		→ LP
O2	listSize	LP	→ int
O3	addLast	LP × string	→ LP
O4	getParam	LP × int	→ string

Pré-conditions : $s \in string$; $i \in int$; $lp \in LP$;

P1 define (getParam (lp, i)) $\Longleftrightarrow$ i > 0 && $\Longleftrightarrow$ i <= listSize (lp)

Axiomes : $s \in string$; $i \in int$; $lp \in LP$;

A1 listSize (createList) = 0

A2 listSize (addLast (lp, s)) = listSize (lp) + 1

A3 getParam (addLast (lp, s), listSize (addLast (lp, s))) = s

Nous définissons le type abstrait de données *ListParameter* pour définir la liste des

paramètres de notre fonction. L'opération *O1* nous permet de connaître la taille de notre liste, elle est définie par les axiomes *A1* et *A2* qui nous permettent de voir qu'une liste vide est de taille 0, et cette taille augmente en fonction du nombre de paramètres. Les opérations *O3* et *O4* sont corrélées car lorsque l'on ajoute un paramètre grâce à *O3*, nous pouvons le récupérer à l'aide de *O4* à la place où nous l'avons inséré (*A3*). Il est à noter que nous ne pouvons récupérer un paramètre que si son index existe, c'est-à-dire qu'il est compris entre 1 et la taille de la liste (*P1*). Nous pouvons maintenant définir le type abstrait *Function* grâce à la liste de paramètres.

TA 10 Function F extends Constraint

Utilise : string, ListString

Opérations

O1	createFunction	string × ListParameter	→ F
O2	getFunction	F	→ string
O3	getParameters	F	→ ListParameter

Axiomes : $f \in string$; $l \in ListString$;

A1 getFunction (createFunction (f, l)) = f

A2 getParamaters (createFunction (f, l)) = l

Nous construisons une fonction à l'aide d'un intitulé et d'une liste de paramètres (*O1*). Nous pouvons retrouver cet intitulé grâce à l'opération *O2* déterminé par le premier paramètre de *O1* (*A1*), et la liste de paramètres par l'opération *O3* en deuxième position (*A2*).

Nous avons donc la possibilité de créer les contraintes que nous pouvons associer ensuite à un *XPath*. Il nous faut maintenant introduire le lien entre un nœud et une contrainte appelée *ConstraintLink*s.

3.4.2.3 Lien de contraintes

Un lien de contraintes ou *ConstraintLink* est un lien permettant de relier un élément à une contrainte, ainsi nous obtenons le TA 11. Cet élément peut être un nœud, une contrainte pour une composition (f *o* g), un autre lien de contrainte pour une double composition (f *o* g *o* h), ou un motif d'arbre d'agrégation.

L'opération de création d'un *ConstraintLink* (*O1*) nécessite un *nœud*, une *Constraint* et un entier. Nous pouvons définir le type de *Constraint* grâce à l'opération *O2* et l'axiome *A1*. De même, le *nœud* est défini par *O3* et *A2*. Enfin, l'indice de la contrainte est déterminé par l'opération *O4* et l'axiome *A3*. Nous pouvons alors enrichir le type abstrait de données *nœud* avec les deux opérations : *getConstraintLink* (*O5*) et *getMaxConstraintLink* (*O6*).

L'opération *O5* nous permet de récupérer un *ConstraintLink* à l'aide de son indice de contrainte défini lors de la création de celui-ci (*A4*). De plus, nous ne pouvons récupérer cet *ConstraintLink* que s'il est défini. Son numéro doit être contenu entre 1 et le nombre maximum de *ConstraintLink* de ce *nœud* (*P1*).

TA 11 ConstraintLink CL

Utilise : Node N, Constraint C, TreePattern TP, Δ = (N $\cup$ C $\cup$ CL $\cup$ TP), Constraint C, XP, int, boolean

Opérations

O1	createConstraintLink	$\Delta \times$ C $\times$ i	$\rightarrow$ CL
O2	getConstraint	CL	$\rightarrow \Delta$
O3	getConstraintElement	CL	$\rightarrow$ N
O4	getConstraintNumber	CL	$\rightarrow$ int
	`opérations pour TA Node`		
O5	getConstraintLink	N $\times$ int	$\rightarrow$ CL
O6	getMaxConstraintLink	N	$\rightarrow$ int
	`opérations pour TA` *`TreePattern`*		
O7	addConstraintToTreePattern	TP $\times$ XP $\times$ boolean $\times$ C	$\rightarrow$ TP
	`opérations pour TA` *`Constraint`*		
O8	getConstraintLink	C	$\rightarrow$ CL

Pré-conditions : $n \epsilon N$; $s \epsilon string$; $i \epsilon int$; $c \epsilon C$;

P1 define (getConstraintLink (n, i)) $\Longleftrightarrow$ i > 0
&& i <= getMaxConstraintLink (e)

P2 define (getConstraintLink (n, i)) $\Longleftrightarrow$ getConstraint (
createConstraintLink (n, c, i)) = c

Axiomes : $\delta \epsilon \Delta$; $c \epsilon C$; $tp \epsilon TP$; $xp \epsilon XP$; $cl \epsilon CL$; $n \epsilon N$; $i \epsilon int$; $b \epsilon boolean$; $s \epsilon String$;

A1 getConstraint (createConstraintLink (δ, c, i)) = c

A2 getConstraintElement (createConstraintLink (δ, c, i)) = δ

A3 getConstraintNumber (createConstraintLink (δ, c, i)) = i

`opérations pour TA` *`nœud`*

A4 getConstraintLink (n, getConstraintNumber (cl)) = cl

A5 getMaxConstraintLink (createNode (s))) = 0

A6 getMaxConstraintLink (getConstraintElement (
createConstraintLink (n, c, i))) = getMaxConstraintLink (n) + 1

`axiomes pour TA` *`TreePattern`*

A7 addConstraintToTreePattern (tp, xp, b, c) =
addXPathToTreePattern (tp, xp, b)

A8 getNode (addConstraintToTreePattern (tp, xp, b, c), xp)
= getConstraintElement (createConstraintLink (getNode (tp, xp), c,
getMaxConstraintLink (getNode (tp, xp))))

A9 getConstraintLink (getConstraint (createConstraintLink (n, c, i)))
= createConstraintLink (n, c, i)

L'opération *O6* nous permet de connaître le nombre maximum de *ConstraintLink* défini pour un *nœud*. Il est de zéro lors de la création du *nœud* (*A5*). Il est incrémenté à chaque création de *ConstraintLink* (*A6*).

Nous pouvons ajouter une opération au *Type Abstrait TreePattern*. En effet, il nous faut une opération d'ajout d'une *Constraint* dans un *TreePattern* que nous associons à un *XPath*. Pour ce faire, nous ajoutons l'opération *O7* : *addConstraintToTreePattern*. Ainsi, grâce à l'axiome *A7*, le *TreePattern* généré par *O7* avec *xp* sur *tp* est le même que celui généré par l'*XPath xp* sur *tp* sans la contrainte. L'axiome *A8* nous permet d'associer le *Type Abstrait Constraint* au *Type Abstrait nœud p* créé par l'*XPath xp* sur *tp* en créant le *ConstraintLink* avec *p* (*getNode (tp, xp)*) et le nombre maximum de *ConstraintLink* déjà existant sur *p* (*getMaxConstraintLink (getNode (tp, xp))*).

Pour finir, l'opération *O8* nous permet de connaître le *ConstraintLink* correspondant à un *Type Abstrait Constraint*, ce lien est déterminé par l'axiome *A9* et par la pré-condition *P2* qui spécifie qu'une contrainte ne peut être récupérée par *O8* que si elle a été attachée à un *nœud* précédemment.

3.4.2.4 Conclusion

Les contraintes maintenant formalisées pourront être associées aux nœuds pour restreindre le domaine auquel elles sont liées. Ces contraintes pourront être utilisées dans certains types d'hyperliens que nous allons maintenant formaliser.

3.4.3 Hyperliens

Les hyperliens que nous avons introduit dans la section 3.3.3 permettent de représenter les différentes opérations possibles dans une requête *XQuery*. Deux classes ont été définies : les hyperliens associatifs et les hyperliens directionnels. Les *AssociationHyperlink* représentent une association restrictive entre plusieurs éléments. Les *DirectionalHyperlink* représentent une transformation d'ensembles pour en obtenir un nouveau. Nous allons donc définir un premier type abstrait de données *Hyperlink* pour que chaque type puisse hériter du même élément.

TA 12 Hyperlink HY

Utilise : TP, int

Opérations

O1	createHyperlink		→ HY
O2	addHyperlink	TP × int × HY	→ TP
O3	getHyperlink	TP × int	→ HY
O4	getSizeHyperlink	TP	→ int

pré-conditions : $tp \in TP$; $i \in int$;

P1 define (getHyperlink (tp, i)) $\Longleftrightarrow$ i > 0 && i <= getSizeHyperlink (tp)

Axiomes : $tp \in TP$; $h \in HY$; $xp \in XP$; $i \in int$; $s \in String$;

A1 getHyperlink (addHyperlink (tp, i, h), i) = h

A2 getSizeHyperlink (createTreePattern (s, xp)) = 0

A3 getSizeHyperlink (addHyperlink (tp, i, h)) = getSizeHyperlink (tp) + 1

Comme nous pouvons le voir dans le TA 12, un hyperlien peut être ajouté à un *TreePattern* par l'intermédiaire de l'opération *O2*, pour cela il nous faut un hyperlien et un indice pour le rajouter (Axiome *A1*). Cet indice doit être compris entre 0 et le nombre maximum d'hyperliens contenus dans un *TreePattern* (pré-conditions *P1*). L'hyperlien peut être récupéré par son identifiant grâce à l'opération *O3* et déterminé par l'axiome *A1*. Le nombre d'hyperliens liés à un *TreePattern* est déterminé par l'opération *O4*. Sa valeur est de zéro lors de la création du *TreePattern* (*A2*). Elle est incrémentée de un à chaque ajout d'hyperlien (*A3*).

Nous avons défini l'ajout des hyperliens dans un *TreePattern*, nous allons maintenant étudier les hyperliens directionnels et les hyperliens associatifs.

3.4.3.1 Hyperliens associatifs

Les hyperliens associatifs (ou *AssociationHyperlink*) sont des hyperliens reliant sous conditions deux éléments du *TGV*. Nous pouvons distinguer deux classes d'associations : *JoinHyperlink* et *ConstraintHyperlink*. Chacun permet de représenter une association particulière du langage *XQuery*.

Hyperlien de jointure

Un hyperlien de jointure (ou *JoinHyperlink*) permet de représenter un lien comparatif entre deux nœuds ou fonctions (*Constraint* ou *ConstraintLink*). Nous pouvons ainsi représenter une jointure entre deux *TreePattern*s au sein d'un même *TGV*. En *XQuery*, nous exprimerions cette jointure par un prédicat liant deux expressions (*XPath*, fonctions sur *XPath*) dans la clause *where* de la requête.

TA 13 JoinHyperlink JH extends Hyperlink

Utilise : Constraint, E ϵ {Node, ConstraintLink, Constraint}

Opérations

O1	createJoinHyperlink	E × E × C	→ HJ
O2	leftJH	HJ	→ E
O3	rightJH	HJ	→ E
O4	getConstraintJH	HJ	→ C

Axiomes : $e_1, e_2 \epsilon E$; $c \epsilon C$;

A1 leftJH (createJoinHyperlink (e_1, e_2, c)) = e_1

A2 rightJH (createJoinHyperlink (e_1, e_2, c)) = e_2

A3 getConstraintJH (createJoinHyperlink (e_1, e_2, c)) = c

Nous pouvons donc créer un hyperlien de jointure grâce à l'opération *O1* à l'aide de deux éléments et d'une contrainte. Nous pouvons définir les deux éléments liés grâce aux opérations *O2* et *O3*, ainsi qu'à leurs axiomes respectifs *A1* et *A2*. la contrainte de comparaison est déterminée grâce à l'opération *O4* et l'axiome *A3*.

Hyperliens de contraintes

Les contraintes de la clause *where* d'une requête *XQuery* sont reliées entre elles par un opérateur booléen (*and/or*). De plus, une contrainte est déterminée à un niveau d'imbrication qui n'est pas toujours identique avec celui de sa variable de déclaration. Ainsi, un hyperlien de contrainte défini l'arborescence des opérateur booléen reliant les contraintes les unes aux autres. Cet hyperlien de contrainte est relié à un *ReturnTreePattern*, permettant de connaître le niveau d'imbrication des contraintes.

Nous pouvons créer ce *ConstraintHyperlink* grâce à l'opération *O1*. La première opération est définie par l'opération *O2* et l'axiome *A1*. De même, la seconde opération est définie par *O3* et *A2*. Nous pouvons connaître l'opération de comparaison grâce à *O4* et *A3*. Ce lien de comparaison est de type booléen : vrai pour un opérateur *and* et faux pour un opérateur *or*.

TA 14 ConstraintHyperlink CH extends Hyperlink

Utilise : E ϵ {C, CH}, C, RTP, boolean

Opérations

O1	createConstraintHyperlink	E × E × boolean	→ CH
O2	leftCH	CH	→ E
O3	rightCH	CH	→ E
O4	getConnectorCH	CH	→ boolean
	`opérations pour TA ReturnTreePattern`		
O5	addConstraintHyperlinkToRTP	RTP × CH	→ RTP
O6	getConstraintHyperlink	RTP	→ CH
O7	getConstraintHyperlinkRTP	CH	→ RTP

Pré-conditions : $rtp \epsilon RTP$; $ch \epsilon CH$;

P1 define (getConstraintHyperlink (rtp))
$\Longleftrightarrow$ addConstraintHyperlinkToRTP (rtp, ch)

P2 define (getConstraintHyperlinkRTP (ch))
$\Longleftrightarrow$ addConstraintHyperlinkToRTP (rtp, ch)

Axiomes : $e_1, e_2 \epsilon E$; $b \epsilon boolean$;

A1 leftCH (createConstraintHyperlink (e_1, e_2, b)) = e_1

A2 rightCH (createConstraintHyperlink (e_1, e_2, b)) = e_2

A3 getConnectorCH (createConstraintHyperlink (e_1, e_2, b)) = b

A4 getConstraintHyperlink (addConstraintHyperlinkToRTP (rtp, ch)) = ch

A5 getConstraintHyperlinkRTP (getConstraintHyperlink (
addConstraintHyperlinkToRTP (rtp, ch))) = rtp

Pour finir, un hyperlien de contrainte peut être associé à un motif d'arbre résultat grâce à l'opération *O5*. Le motif d'arbre est défini par l'opération *O6* et l'axiome *A4* et l'hyperlien de contrainte par *O7* et *A5*. Les pré conditions *P1* et *P2* permettent de définir que les opérations *O6* et *O7* uniquement quand un hyperlien de contrainte est associé à un motif d'arbre résultat.

3.4.3.2 Hyperliens Directionnels

Les *hyperliens directionnels* (ou *DirectionalHyperlink*) sont des *Hyperlink*s orientés. Ils permettent de définir une relation entre deux éléments. Nous pouvons distinguer trois types d'hyperliens directionnels : *ProjectionHyperlink*, *SetHyperlink*, *GeneralizedHyperlink*, *ExplorationHyperlink* et *IfThenElseHyperlink* que nous étudions ci-dessous.

Les hyperliens directionnels sont des injections dans nos TGVs. Ce qui veut dire que pour chaque élément ciblé, il n'existe qu'un seul élément de projection. Ceci se traduira par une pré-condition dans le *Type Abstrait DirectionalHyperlink* (TA 15). De plus, ces hyperliens peuvent être optionnels au même titre que les *NodeLink*s.

Nous pouvons voir que le type abstrait de données *DirectionalHyperlink* utilise le type E, celui-ci représente le couple de type abstrait $Node$ ou $TreePattern$. Ceux-ci seront définis dans les hyperliens que nous étudions par la suite. Le premier élément de l'hyperlien directionnel est défini par l'opération *O2* et l'axiome (*A1*). Le second élément est la destination de cet hyperlien, il est défini par l'opération *O3* et l'axiome

TA 15 DirectionalHyperlink DH extends Hyperlink

Utilise : E ϵ {Node N, TreePattern TP}, boolean

Opérations

O1	createDirectionalHyperlink	E × E × boolean	→ DH
O2	fromDH	DH	→ E
O3	toDH	DH	→ E
O4	optionalPH	DH	→ boolean

Pré-conditions : $\forall\ dh_1, dh_2\ \epsilon\ DH$;

P1 toDH (dh_1) != toDH (dh_2)

Axiomes : $e_1, e_2\ \epsilon\ E$; $b\ \epsilon\ boolean$;

A1 fromDH (createDirectionalHyperlink $(e_1, e_2, b)) = e_1$

A2 toDH (createDirectionalHyperlink $(e_1, e_2, b)) = e_2$

A3 optionalDH (createDirectionalHyperlink $(e_1, e_2, b)) = b$

A2. L'optionalité de l'hyperlien directionnel est définie par l'opération *O4* et l'axiome *A3*. Cela nous permet de déterminer si ce lien est obligatoire lors de l'évaluation du TGV ou si le résultat est optionnel. Comme les hyperliens directionnels sont des injections, l'élément projeté est unique. Ainsi, nous ne pouvons avoir deux hyperliens directionnels liés à un même nœud de destination (*toDH*). Cette règle est induite par la pré-condition *P1*.

Maintenant que nous avons le type générique pour définir un hyperlien directionnel, nous allons étudier les différents types d'hyperliens directionnels nous permettant de représenter les notions spécifiques du langage *XQuery* : Projection, Spécialisation, Généralisation, Ensembliste et Conditionnel.

Hyperliens de projection

Un hyperlien de projection (ou *ProjectionHyperlink*) est un lien de projection d'un nœud vers un autre nœud. Cet hyperlien permet de projeter la valeur du premier nœud vers le second. Cet *Hyperlink* sera très utile pour la création de la clause de reconstruction que nous étudions dans le *Type Abstrait ReturnTreePattern* (section 3.4.4.2).

TA 16 ProjectionHyperlink PH extends DirectionalHyperlink

Utilise : Node N, boolean

Opérations

O1	createProjectionHyperlink	N × N × boolean	→ PH

Axiomes : $n_1, n_2\ \epsilon\ N$; $b\ \epsilon\ boolean$;

A1 fromDH (createProjectionHyperlink $(n_1, n_2, b)) = n_1$

A2 toDH (createProjectionHyperlink $(n_1, n_2, b)) = n_2$

A3 optionalDH (createProjectionHyperlink $(n_1, n_2, b)) = b$

Grâce à un *ProjectionHyperlink* nous pouvons créer un lien entre deux nœuds (*O1*). Les types utilisés sont donc deux nœuds qui peuvent donc être déterminés par les opérations $fromDH$ et $toDH$ grâce aux axiomes respectifs *A1* et *A2*. L'optionalité de l'hyperlien de projection est déterminé par l'axiome *A3*.

Hyperliens d'exploration

L'hyperlien d'exploration (ou *ExplorationHyperlink*) représente l'exploration d'un élément d'un *TreePattern*. Par exemple, dans le langage XQuery, il est possible de définir une nouvelle variable de définition sur un *XPath* (par exemple dans une requête imbriquée). Nous pouvons, sur cette nouvelle variable, définir un ensemble de contraintes. Pour les TGV, nous traduisons cela par un nouveau motif d'arbre qui explore celui dont il provient. Il nous faut donc un lien entre le nœud du premier motif d'arbre vers le second défini par une variable. Ainsi, l'hyperlien d'exploration nous permet de définir ce lien entre un nœud vers un motif d'arbre.

TA 17 ExplorationHyperlink EH extends DirectionalHyperlink

Utilise : Node N, TreePattern TP, boolean

Opérations

O1 createExplorationHyperlink N × TP × boolean → EH

Axiomes : $n \in N$; $tp \in TP$; $b \in boolean$;

A1 fromDH (createExplorationHyperlink (n, tp, b)) = n

A2 toDH (createExplorationHyperlink (n, tp, b)) = tp

A3 optionalDH (createExplorationHyperlink (n, tp, b)) = b

Un *ExplorationHyperlink* se construit à l'aide d'un élément $Node$ et d'un élément TP (*O1*). Les types spécifiques de cet hyperlien sont donc déterminés pour les opérations $fromDH$ et $toDH$ grâce aux axiomes respectifs *A1* et *A2*. Un lien de spécialisation peut être optionnel, celui-ci est déterminé grâce à l'axiome *A3*. Cela nous permet de définir que le *TreePattern* est optionnel par rapport au second nœud grâce au *ExplorationHyperlink*.

Hyperliens de généralisation

Un hyperlien de généralisation (ou *GeneralizedHyperlink*) permet de représenter la généralisation d'un motif d'arbre (généralement ATP). Cela permet de définir la projection du résultat d'un motif d'arbre vers un nœud d'un autre motif. Dans le langage *XQuery*, nous pourrions prendre l'exemple du *let* que nous projetterions dans la clause *return* (après l'utilisation de la canonisation de requêtes, section 2.2.3).

TA 18 GeneralizedHyperlink GH extends DirectionalHyperlink

Utilise : Node N, TreePattern TP, boolean

Opérations

O1 createGeneralizedHyperlink TP × N × boolean → GH

Axiomes : $n \in N$; $tp \in TP$; $b \in boolean$;

A1 fromDH (createGeneralizedHyperlink (tp, n, b)) = tp

A2 toDH (createGeneralizedHyperlink (tp, n, b)) = n

A3 optionalDH (createGeneralizedHyperlink (tp, n, b)) = b

Le *Type Abstrait GeneralizedHyperlink* se construit à l'aide d'un élément TP et d'un éléménet $Node$ (*O1*). Les types spécifiques de l'hyperlien de généralisation déterminent les opérations $fromDH$ et $toDH$ grâce aux axiomes *A1* et *A2*.
L'optionalité du motif d'arbre d'origine, par rapport au nœud de destination, est définie par à l'axiome *A3*.

Hyperliens ensemblistes

Comme nous avons pu le voir dans la section 3.3.3, les hyperliens ensemblistes (ou *SetHyperlink*) sont des hyperliens directionnels permettant de relier plusieurs motifs d'arbres pour en produire un nouveau qui est projeté sur un nœud, de la même manière que les hyperliens de généralisation. L'opération ensembliste sera définie sous forme d'une contrainte à l'aide de sa définition : «*union*», «*intersect*» et «*except*».

Pour ce type abstrait de données, nous considérons le type *(TP)* comme une liste de motifs d'arbre devant contenir au moins deux motifs d'arbre. L'opération principale est *getTP (l, i)* permettant de récupérer un motif d'arbre de la liste à la position i. La seconde opération, *getSizeTP (l)*, correspond à la taille de cette liste retournant un entier.

TA 19 SetHyperlink SH extends Hyperlink

Utilise : Node N, TreePattern TP, Constraint C, String, int

Opérations

O1	createSetHyperlink	TP × N × C	→ SH
O2	getTPSH	SH × int	→ TP
O3	getSizeTPSH	SH	→ int
O4	resultSH	SH	→ N
O5	getConstraintSH	SH	→ C

Pré-conditions : $l \in TP$; $i \in int$;

P1 define (getTPSH (l, i)) ⇔ i > 0 && i <= getSizeTPSH (l)

Axiomes : $l \in (TP)$; $n \in N$; $c \in C$; $i \in int$;

A1 getTPSH (createSetHyperlink (l, n, c), i) = getTP (l, i)

A2 getSizeTPSH (createSetHyperlink (l, n, c)) = getSizeTP (l)

A3 resultSH (createSetHyperlink (l, n, c)) = n

A4 getConstraintSH (createSetHyperlink (l, n, c)) = c

L'hyperlien ensembliste est créé grâce à l'opération *O1* prenant la liste des *TreePatterns* utilisés (TP), un nœud de projection et une opération ensembliste. Chaque motif d'arbre de (TP) peut-être récupéré grâce à l'opération *O2* (*A1*) dont l'indice doit être compris entre 0 et la taille de la liste (*O3*, *A2* et *P1*). Le nœud de projection est défini par l'opération *O4* et l'axiome *A3*. La contrainte ensembliste est déterminée par l'opération *O5* et son axiome *A4*.

Ainsi, l'hyperlien ensembliste projette son résultat sur le nœud défini par l'opération *O4*. L'opération ensembliste de ce lien dépend de sa contrainte définie par *O5* qui traitera les données provenant de l'ensemble des domaines déterminés par *O2* et *O3*.

Hyperliens conditionnels

Dans le langage *XQuery*, il existe des expressions conditionnelles de type *if <expr> then <expr> else <expr>*. Pour représenter cela, nous avons besoin d'une expression de test (if), deux expressions pour la clause *then* et la clause *else*, et enfin un nœud de projection équivalente à un *GeneralizedHyperlink*. Les clauses *then* et*else* proviennent d'un *XPath* ou d'un *let* (voir canonisation 3.2), ainsi nous utilisons un nœud ou un motif d'arbre pour les définir. Les hyperliens conditionnels sont définis

par le *Type Abstrait IfThenElseHyperlink*.

TA 20 IfThenElseHyperlink IH extends Hyperlink

Utilise : Node N, Constraint C, E ϵ {N, TP}

Opérations

O1	createIfThenElseHyperlink	C × E × E × N	→ IH
O2	ifIH	IH	→ C
O3	thenIH	IH	→ E
O4	elseIH	IH	→ E
O5	resultIH	IH	→ N

Axiomes : $c \epsilon C$; $e_1, e_2 \epsilon E$; $n \epsilon N$;

A1 ifIH (createIfThenElseHyperlink $(c, e_1, e_2, n)) = c$

A2 thenIH (createIfThenElseHyperlink $(c, e_1, e_2, n)) = e_1$

A3 elseIH (createIfThenElseHyperlink $(c, e_1, e_2, n)) = e_2$

A4 resultIH (createIfThenElseHyperlink $(c, e_1, e_2, n)) = n$

L'opération de création *O1* détermine chacun des éléments nécessaires au *IfThenElseHyperlink*. La contrainte conditionnelle est définie par l'opération *O2* et l'axiome *A1*. Les deux éléments potentiels de type E ($Node$ ou $TreePattern$) sont définis par *O3* et *O4* et leurs axiomes respectifs *A2* et *A3*. Le nœud de projection du résultat est déterminé par l'opération *O5* et l'axiome *A4*.

Ainsi, si la contrainte déterminée par l'opération *O2* est vraie pour une instance de données, alors l'élément défini par l'opération *O3* sera projeté sur le nœud résultat déterminé par l'opération *O5*. Sinon, l'élément défini par l'opération *O4* sera projeté sur ce même nœud.

3.4.3.3 Conclusion

Nous avons ainsi formalisé les hyperliens qui permettent de relier les différents éléments composant les TGV. Les hyperliens associatifs restreignent les ensembles résultats à l'aide de contraintes, ceux-ci sont produits par les expressions provenant de la clause *where*. Les hyperliens directionnels produisent de nouveaux ensembles d'arbres à l'aide de projections, ceux-ci proviennent des déclarations de variables, des *XPath* présents dans la clause *return* et des opérations ensemblistes et conditionnelles. Nous allons maintenant définir les différents types de motifs d'arbres reliés par les hyperliens.

3.4.4 Les Différents Motifs d'Arbre

Comme nous avons pu le voir jusqu'ici, les TGVs sont un ensemble de *TreePatterns* contenant des arbres de nœuds, chacun d'eux reliés par des *Hyperlink*s. Le langage de requête *XQuery* étant riche, nous devons définir les différentes spécifications des variables et les clauses de construction de résultats. Il va donc nous falloir introduire de nouveaux types de *TreePatterns* pour répondre à nos besoins. Ainsi,

nous allons étudier les *Types Abstraits* suivants : *SourceTreePattern*, *IntermediateTreePattern*, *ReturnTreePattern* et *AggregateTreePattern*, dont les caractéristiques d'héritage peuvent être représentées dans le diagramme de la figure 3.13. Les trois types *SourceTreePattern*, *IntermediateTreePattern* et *ReturnTreePattern* héritent des *TreePatterns*, alors que le *Type Abstrait AggregateTreePattern* hérite des caractéristiques du *ReturnTreePattern*.

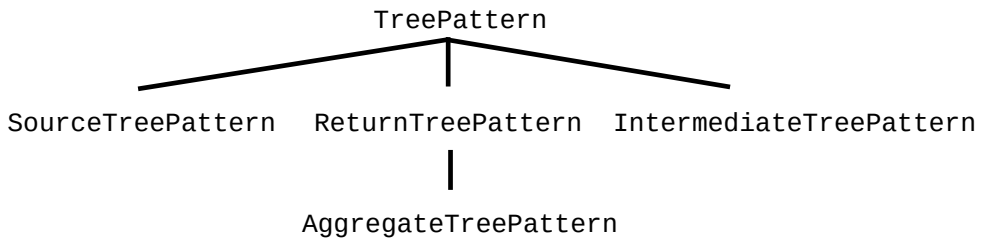


FIG. 3.13 – Diagramme d'héritage des *TreePatterns*

3.4.4.1 SourceTreePattern

Le *SourceTreePattern* (ou *STP*) est un *TreePattern* qui permet de définir un motif d'arbre de sélection pour des documents XML. Nous devons définir la source du *TreePattern* par le document cible, et la racine par l'*XPath* de définition du domaine (ie. for $i in collection("catalog")/catalog/book). Dans le cas où aucun document n'est défini (ie. for $i in /catalog/book), il faut alors choisir la source "*" (correspondant à l'ensemble des collections).

TA 21 SourceTreePattern extends TreePattern STP

Utilise : String, XP, P

Operations

O1	createSTP	String × String × XP	→ STP
O2	getSourceSTP	STP	→ N

Axiomes : $v, s \in String$; $xp \in XP$;

A1 getVariable (createSTP (v, s, xp)) = v

A2 getSourceSTP (createSTP (v, s, xp)) = createNode (s)

L'opération *O1* permet de définir un *SourceTreePattern*. La variable de ce motif d'arbre est définie par l'opération *getVariable*, provenant du type *TreePattern* (dont STP étend). L'axiome *A1* permet de définir d'où provient la variable lors de la création. La source étant la collection, elle est déterminée par l'opération *O2* et l'axiome *A2* qui nous permet de voir qu'un *nœud* est créé à partir de la chaîne de caractères donnée à la création. Les opérations et axiomes provenant du TA 6 restent inchangés, tel que l'ajout de l'*XPath* se rajoute sur la source, et le dernier élément donne la racine. Le reste se comporte donc comme un *TreePattern*.

3.4.4.2 ReturnTreePattern

Un *ReturnTreePattern* (ou *RTP*) est un *TreePattern* dont les nœuds représentent les balises à construire pour le document XML résultat. Un *nœud* se projette dans le *SourceTreePattern* à l'aide d'un *ProjectionHyperlink* et un *TreePattern* avec un *GeneralizedHyperlink*, ces hyperliens permet de bâtir le résultat demandé dans la clause de reconstruction de résultats. Nous pouvons ajouter que chacun des *NodeLink*s d'un *SourceTreePattern* ont un axe type '*child*'. En effet, il n'existe aucun autre type d'axe entre les balises d'un document XML. De plus, ces *NodeLink*s sont obligatoires car la balise est nécessaire à la construction du résultat.

TA 22 ReturnTreePattern extends TreePattern RTP

Utilise : String, XP

Operations

O1	createRTP	String	→ RTP
O2	addXPathToRTP	RTP × XP	→ RTP
	`Opération à ajouter au TA XP`		
O3	isAllChildren	XP	→ boolean

Pré-conditions : $rtp \in RTP$; $xp \in XP$;

P1 define (addXPathToRTP (rtp, xp)) $\Longleftrightarrow$ isAllChildren (xp) = true

Axiomes : $s \in String$; $i \in int$; $p \in P$; $xp \in XP$;

A1 getVariable (createRTP (s)) = s

A2 getRoot (createRTP (s)) = createPattern (s)

A3 getSource (createRTP (s)) = createPattern (s)

A4 isEmpty (xp) ⇒ isAllChildren (xp) = true

A5 isAllChildren (xp) ⇒ getAxis (first (xp)) = 'child'
&& isAllChildren (deleteFirst (xp))

A6 addXPathRTP (rtp, xp) = addXPathToTreePattern (rtp, xp, true)

Nous pouvons créer un *ReturnTreePattern* à l'aide d'une chaîne de caractère grâce à l'opération *O1*. Cette variable permet de définir la variable de déclaration (*A1*), la racine (*A2*) et la source (*A3*) de ce *TreePattern* avec cette chaîne de caractère. Ainsi, nous obtenons un *ReturnTreePattern* auquel nous pouvons ajouter des *XPath*s.
Pour ajouter un *XPath* à un *ReturnTreePattern*, il faut vérifier que l'*XPath* est composé exclusivement de liens de type parents/enfants (*P1*). En effet, les liens du *ReturnTreePattern* ne doivent pas posséder de liens ancêtres/descendants, ce qui est vérifié par l'opération *O3* et les axiomes *A4* et *A5* qui testent chacun des liens de l'*XPath*.
Pour finir, nous pouvons ajouter chaque *XPath* (*O2*) de manière obligatoire dans le *ReturnTreePattern*, ce qui est impliqué par l'axiome *A6* qui reprend l'opération *O8* du *Type Abstrait TP* avec la valeur «*true*».
Afin d'intégrer les éléments textuels à notre représentation, nous identifions ces *nœud*s par une quote ' " ' entourant la chaîne de caractère à identifier dans la clause *return*.

Les opérations nous permettant de relier les hyperliens *ProjectionHyperlink*, *ExplorationHyperlink* et *IfThenElseHyperlink* au *ReturnTreePattern* sont détaillées dans le *Type Abstrait TGV*.

3.4.4.3 IntermediateTreePattern

Un *IntermediateTreePattern* (ou *ITP*) correspond à la définition d'une spécialisation d'un nœud défini dans un autre *TreePattern*. La source de ce *TreePattern* correspond donc à l'élément projeté, ainsi nous devons copier le label du *nœud* du *TreePattern* d'origine pour créer la source du nouvel *IntermediateTreePattern*. Comme pour le *TreePattern*, la racine de cet *ITP* est déterminé par un *XPath* défini par la déclaration de la variable dans la requête *XQuery*. De plus, nous pouvons ajouter qu'il existe donc un *ExplorationHyperlink* entre le *nœud* d'origine et le nœud source de l'*IntermediateTreePattern*.

TA 23 IntermediateTreePattern extends TreePattern ITP
Utilise : String
Opérations
O1 createITP String × String → ITP
Axiomes : $v, r \in String$;
A1 getVariable (createITP (v, r)) = v
A2 getSource (createITP (v, r)) = createNode (r)
A3 getRoot (createITP (v, r)) = getSource (createITP (v, r))

Un *IntermediateTreePattern* se construit à l'aide de deux chaînes de caractères (*O1*). Les axiomes *A1*, *A2* et *A3* permettent de redéfinir les axiomes du *TreePattern* en fonction de la définition d'un *ITP*, ainsi nous pouvons voir que la variable de définition est déterminée par la première chaîne de caractères (*A1*), la source de l'*ITP* est définie par la seconde chaîne de caractères (*A2*) et enfin, la racine est équivalente à la source (*A3*).

3.4.4.4 AggregateTreePattern

Un *AggregateTreePattern* (ou *ATP*) est un *TreePattern* représentant un agrégat de données. Cette opération d'agrégat s'applique à l'ensemble des données filtrées par l'*ATP*. Le contenu de l'*AggregateTreePattern* correspond à une reconstruction de résultats que l'on pourra alors projeter dans le *ReturnTreePattern* par l'intermédiaire d'un *GeneralizedHyperlink*. Cet hyperlien généralise le contenu de l'*ATP*. Nous allons donc introduire le nouveau *Type Abstrait AggregateTreePattern*. La clause *order by* permet de traiter de manière globale les données, de ce fait, cette clause sera représentée par un *AggregateTreePattern* dont la fonction sera un «*order by*».

Comme nous pouvons le constater dans le TA 24, un *AggregateTreePattern* se construit de la même manière qu'un *ReturnTreePattern* à ceci prêt que nous lui associons une contrainte d'agrégation. Cette fonction est définie dans l'opération de création *O1*, ainsi que dans l'opération de définition *O2* déterminée par l'axiome *A4*. Les axiomes *A1*, *A2* et *A3* nous permettent de redéfinir les opérations *getDeclarationName*, *getRoot* et *getSource* par rapport au constructeur de l'*AggregateTreePattern* : *O1*.

TA 24 AggregateTreePattern extends ReturnTreePattern ATP

Utilise : String, C

Operations :

O1	createATP	String × C	→ ATP
O2	getATPConstraint	ATP	→ C

Axiomes : $s \in String$; $c \in C$;

A1 getVariable (createATP (s, c)) = s

A2 getRoot (createATP (s, c)) = createNode (s)

A3 getSource (createATP (s, c)) = createNode (s)

A4 getATPConstraint (createATP (s, c)) = c

3.4.4.5 Conclusion

Nous avons dans cette section étudié les quatre types de motifs d'arbre nous permettant de définir les différents cas possibles d'arbres pour représenter une requête *XQuery*. Les *SourceTreePattern*s nous permettent de définir un domaine basé sur une collection où un ensemble de collections sur lequel nous associons un *TreePattern*. Le *ReturnTreePattern* nous permet de définir la reconstruction de résultat sous forme d'un arbre contenant les balises pour bâtir le document XML, ainsi que des *ProjectionHyperlink*s déterminant les nœuds à projeter pour ce document. Les *IntermediateTreePattern*s définissent de nouveaux sous-domaines à partir d'un *TreePattern* existant, ils se basent sur un *ExplorationHyperlink* entre le nœud d'origine et l'*ITP*. Enfin, les *AggregateTreePattern*s déterminent les opérations d'agrégats et les opérations sur les ensembles résultats, ils sont aussi utilisés pour définir les clauses *let*. Nous étudions maintenant le *Type Abstrait TGV* qui nous permettra de relier les *TreePattern*s et les *Hyperlink*s pour obtenir les représentations générales d'une requête *XQuery*.

3.4.5 Tree Graph View (TGV)

Nous avons jusqu'ici étudié les *TreePattern*s qui nous permettent de définir des domaines et de reconstructions, ainsi que les *Hyperlink*s définissant les liens entre les *TreePattern*s ou les éléments des *TreePattern*s. Nous devons maintenant relier chacune de ces représentations par un *Type Abstrait* global que nous appelons *TGV* ou *TGV*.

Un *TGV* est un type abstrait permettant de relier l'ensemble des éléments qui le compose. Ainsi, grâce à celui-ci, il est possible d'associer des *XPath* à des *TreePattern*, des hyperliens entre les éléments et des contraintes. Il est aussi possible de récupérer ces informations et de les modifier.

a) Construction du TGV

L'opération *O1* nous permet de construire un *TGV* vide, nous pouvons ajouter des *TreePattern* grâce à l'opération *O2* et les récupérer par sa variable de définition

TA 25 TreeGraphView TGV

Use : string, Node N, XPath XP, Hyperlink HY, TreePattern TP

Opérations

O1	createTGV		→ TGV
O2	addTreePattern	TGV × TP	→ TGV
O3	getTreePattern	TGV × string	→ TP
O4	getTGVNode	TGV × XP	→ N
O5	addXPathToTGV	TGV × XP × boolean	→ TGV
O6	addConstraint-ToTGV	TGV × XP × boolean × C	→ TGV
O7	addPH	TGV × XP × XP × boolean	→ TGV
O8	addEH	TGV × XP × string × boolean	→ TGV
O9	addGH	TGV × string × XP × boolean	→ TGV
O10	addJH	TGV × XP × boolean × XP × boolean × C	→ TGV
O11	addSH	TGV × string × string × XP × string	→ TGV
O12	addIH	TGV × XP × C × string × string × XP	→ TGV
O13	addCH	TGV × C × C × boolean	→ TGV

Axiomes : $b \epsilon$ *boolean*; $s \epsilon$ *string*; $xp, xp_1, xp_2 \epsilon XP$; $tp \epsilon TP$;

A1 getVariable (getTreePattern (tgv, s)) = s

A2 getTreePattern (addTreePattern (tgv, tp), getVariable (tp)) = tp

A3 getTGVNode (tgv, xp) = getNode (getTreePattern (
tgv, labelElement (first (xp))), deleteFirst (xp))

A4 getTGVNode (addXPathToTGV (tgv, xp, b), xp) = getPattern
(addXPathToTreePattern (getTreePattern (tgv,
labelElement (first (xp))), deleteFirst (xp), b), deleteFirst (xp))

A5 getTGVNode (addConstraintToTGV (tgv, xp, b, o), xp) = getNode
(addConstraintToTreePattern (getTreePattern (tgv,
labelElement (first (xp))), deleteFirst (xp), b, o), deleteFirst (xp))

grâce à l'opération *O3*. Ils sont déterminés par l'axiome *A1* et *A2*. Un *TreePattern* est déterminé par son nom de variable.

Un *XPath* (optionnel ou non) peut être ajouté sur un *TGV* par l'intermédiaire de l'opération *O5*, cela permet de créer un nœud qui pourra être récupéré suivant son *XPath* dans l'arbre par l'opération *O4*. Les opérations du *TGV* utilisent celles définies dans le *Type Abstrait TreePattern* (TA 6) : *getNode, addXPathToTreePattern*. Comme nous pouvons le voir dans les axiomes *A3* et *A4*, le nom du *TreePattern* sélectionné est défini par le premier élément du *XPath* (*labelElement (first (xp))*), ensuite, le reste de cet *XPath* est ajouté au *TreePattern* (*deleteFirst (xp)*).

L'ajout d'un TA *Constraint* dans le *TGV* est donné par l'opération *O6*. L'axiome associé *A5* détermine que cette contrainte utilise l'opération définie dans le *Type Abstrait ConstraintLink* (TA 11) : *addConstraintToTreePattern*.

Les opérations *O7* à *O13* permettent de définir les différents ajouts d'hyperliens dans un *TGV*. Chacune demandant un certain nombre d'axiomes, nous les étudions séparément.

b) Ajout d'un hyperlien de projection

Axiomes : $s_1, s_2, e \in string$; $xp \in XP$; $tgv \in TGV$;

A6 addPH (tgv, xp_1, xp_2, b)
= addXPathToTGV (addXPathToTGV (tgv, xp_1, b), xp_2, b)

A7 fromDH (createProjectionHyperlink (
getTGVNode (addPH (tgv, xp_1, xp_2, b), xp_1),
getTGVNode (addPH (tgv, xp_1, xp_2, b), xp_2))
= getTGVNode (addPH (tgv, xp_1, xp_2, b), xp_1)

A8 toDH (createProjectionHyperlink (
getTGVNode (addPH (tgv, xp_1, xp_2, b), xp_1),
getTGVNode (addPH (tgv, xp_1, xp_2, b), xp_2))
= getTGVNode (addPH (tgv, xp_1, xp_2, b), xp_2)

A9 optionnalDH (createProjectionHyperlink (
getTGVNode (addPH (tgv, xp_1, xp_2, b), xp_1),
getTGVNode (addPH (tgv, xp_1, xp_2, b), xp_2)) = b

Les hyperliens de projection sont ajoutés au *TGV* grâce à l'opération *O7*. Ils nécessitent deux *XPath*s pour déterminer chacun des nœuds du *ProjectionHyperlink*, ces deux *XPath*s sont ajoutés au *TGV* (*A6*). Comme nous pouvons le voir dans les *A7* et *A8*, le *ProjectionHyperlink* est déterminé par les deux nœuds créés à l'aide des *XPath* donnés en paramètre. L'optionalité de cet hyperlien est définie à sa création grâce à l'axiome *A9*.

c) Ajout d'un hyperlien d'exploration

Axiomes : $b \in boolean$; $s \in string$; $xp \in XP$; $tgv \in TGV$;

A9 addEH (tgv, xp, s, b) = addTreePattern (addXPathToTGV (
tgv, xp, b), createITP (s, labelElement (last (xp))))

A10 fromDH (createExplorationHyperlink (getTGVNode (addEH (
tgv, xp, s, b), xp), getTreePattern (addEH (tgv, xp, s, b), s), b))
= getTGVNode (addEH (tgv, xp, s, b), xp)

A11 toDH (createExplorationHyperlink (getTGVNode (addEH (tgv, xp, s, b), xp),
getTreePattern (addEH (tgv, xp, s, b), s), b))
= getTreePattern (addEH (tgv, xp, s, b), s)

A12 optionnalDH (createExplorationHyperlink (getTGVNode (addEH (
tgv, xp, s, b), xp), getTreePattern (addEH (tgv, xp, s, b), s), b)) = b

Les hyperliens d'exploration sont créés à l'aide de l'opération *O8*, ceux-ci sont déterminés par un *XPath* pour le *Node* d'origine et une chaîne de caractères pour le nom du *TreePattern* cible, et un booléen pour l'optionalité du lien. Lors de la création de cet hyperlien, nous devons initialiser un nœud et un *IntermediateTreePattern* (*A9*), le nœud est créé à l'aide de l'*XPath*, et l'*ITP* grâce au dernier élément de cet *XPath* et de la variable (*createITP (s, labelElement (last (xp)))*). Le nœud d'origine du *ExplorationHyperlink* est déterminé par l'axiome *A10*, le *TreePattern* cible par l'axiome *A11* et l'optionalité de cet hyperlien par l'axiome *A12*.

d) Ajout d'un hyperlien de généralisation

Axiomes : $s_1, s_2, e \in string$; $xp \in XP$; $tgv \in TGV$;

A13 addGH (tgv, xp, s, b) = addXPathToTGV (tgv, xp, b)

A14 fromDH (createGeneralizedHyperlink (getTreePattern (addGH (
tgv, s, xp, b), s),getTGVNode (addGH (tgv, s, xp, b), xp), b))
= getTreePattern (addGH (tgv, s, xp, b), s)

A15 toDH (createGeneralizedHyperlink (getTreePattern (addGH (
tgv, s, xp, b), s),getTGVNode (addGH (tgv, s, xp, b), xp), b))
= getTGVNode (addGH (tgv, s, xp, b), xp)

A16 optionnalDH (createGeneralizedHyperlink (getTreePattern (addGH (
tgv, s, xp, b), s),getTGVNode (addGH (tgv, s, xp, b), xp), b)) = b

Les hyperliens de généralisation sont déterminés par l'opération *O9* qui prend en paramètre un *XPath*, une chaîne de caractères et un booléen. l'opération *A9* ajoute l'*XPath* dans le *TreePattern* déterminé par l'axiome *A13*, ce qui nous donne un nœud cible pour notre *GeneralizedHyperlink*. l'axiome *A14* nous permet de déterminer le *TreePattern* source de cet hyperlien grâce à sa variable, alors que l'axiome *A15* nous donne le *Node* cible créé par l'opération *O9*. Enfin, l'axiome *A16* nous donne l'optionalité de l'hyperlien de généralisation donnée par le booléen à la création.

e) Ajout d'un hyperlien de jointure

Axiomes : $c \in C$; $b_1, b_2 \in boolean$; $xp_1, xp_2 \in XP$; $tgv \in TGV$;

A17 addJH (tgv, xp_1, b_1, xp_2, b_2, c)
addXPathToTGV (addXPathToTGV (tgv, xp_1, b_1), xp_2, b_2)

A18 leftJH (createJoinHyperlink (
getTGVNode (addJH (tgv, xp_1, b_1, xp_2, b_2, c), xp_1),
getTGVNode (addJH (tgv, xp_1, b_1, xp_2, b_2, c), xp_2), c))
= getTGVNode (addJH (tgv, xp_1, b_1, xp_2, b_2, c), xp_1)

A19 rightJH (createJoinHyperlink (
getTGVNode (addJH (tgv, xp_1, b_1, xp_2, b_2, c), xp_1),
getTGVNode (addJH (tgv, xp_1, b_1, xp_2, b_2, c), xp_2), c))
= getTGVNode (addJH (tgv, xp_1, b_1, xp_2, b_2, s), xp_2)

A20 getConstraint (createJoinHyperlink (
getTGVNode (addJH (tgv, xp_1, b_1, xp_2, b_2, s), xp_1),
getTGVNode (addJH (tgv, xp_1, b_1, xp_2, b_2, s), xp_2), c))
= c

Les hyperliens de jointure sont déterminés par l'opération *O10*, grâce à deux *XPath*s, deux valeurs booléennes et une chaîne de caractères. Chaque *XPath* associé à un booléen donnera un *Node*, tous deux reliés par un *JoinHyperlink* avec une contrainte déterminée par la chaîne de caractères. Ainsi, l'axiome *A17* détermine les deux *XPath*s à ajouter au *TGV*. L'axiome *A18* défini le nœud de gauche (*leftJH*) par le premier *XPath*, alors que le nœud de droite (*rightJH*) sera défini par le second *XPath*. L'opération du *JoinHyperlink* est déterminé par la chaîne de caractères *s* (*createConstraint (s)*) dans l'axiome *A20*.

f) Ajout d'un hyperlien ensembliste

Axiomes : $s_1, s_2, e \in string$; $xp \in XP$; $tgv \in TGV$;

A21 addSH (tgv, s_1, s_2, xp, e) = addXPathToTGV (tgv, xp, true)

A22 firstSH (createSetHyperlink (
getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_1),
getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_2),
getTGVNode (addSH (tgv, s_1, s_2, xp, e), xp), e))
= getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_1)

A23 secondSH (createSetHyperlink (
getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_1),
getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_2),
getTGVNode (addSH (tgv, s_1, s_2, xp, e), xp), e))
= getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_2)

A24 resultSH (createSetHyperlink (
getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_1),
getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_2),
getTGVNode (addSH (tgv, s_1, s_2, xp, e), xp), e))
= getTGVNode (addSH (tgv, s_1, s_2, xp, e), xp)

A25 getConstraintSH (createSetHyperlink (
getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_1),
getTreePattern (addSH (tgv, s_1, s_2, xp, e), s_2),
getTGVNode (addSH (tgv, s_1, s_2, xp, e), xp), e)) = e

Les hyperliens ensemblistes sont déterminés par l'opération *O11* qui demande deux chaînes de caractères pour les deux *TreePattern*s d'origine, un *XPath* pour le nœud cible, et une troisième chaîne de caractères pour donner l'opération ensembliste reliant les trois éléments. L'axiome *A21* nous montre que l'opération *O11* ajoute un *XPath* dans le *TGV*. Cet *XPath* est utilisé pour l'axiome *A24* pour déterminer le nœud cible (*resultSH*). Le premier *TreePattern* (*firstSH*) déterminé par l'axiome *A22* est défini par la chaîne de caractères s_1 (*getTreePattern (addSH (tgv,* s_1*,* s_2*, xp, e),* s_1*)*). Le second *TreePattern* (*secondSH*) déterminé par l'axiome *A23* est défini par la chaîne de caractères s_2 (*getTreePattern (addSH (tgv,* s_1*,* s_2*, xp, e),* s_2*)*). Enfin, l'opération ensembliste est déterminée par l'axiome *A25* à partir de la chaîne de caractères "*e*".

g) Ajout d'un hyperlien conditionnel

Les hyperliens conditionnels *IfThenElse* peuvent être ajoutés au *TGV* grâce à l'opération *O12* "*AddIH*" qui prend en paramètre un *XPath* et une contrainte pour la condition, deux chaînes de caractères pour sélectionner les deux *TreePattern*s sources, un deuxième *XPath* permet de produire le nœud de destination (*A26*). L'axiome *A27* nous permet de définir la contrainte de test ("*ifIH*") à partir de la variable *c* lors de la création. L'axiome *A28* nous permet de définir le *TreePattern* à projeter si le test est *vrai* ("*thenIH*"), sinon, "*elseIH*" est utilisé grâce à l'axiome *A29*. Les deux *TreePattern*s sont projetés sur un nœud défini grâce à l'axiome *A30*, qui est donc déterminé par l'*XPath* xp_2.

Axiomes : $s_1, s_2, e \in string$; $xp \in XP$; $tgv \in TGV$; $c \in C$;

A26 addIH (tgv, xp_1, c, s_1, s_2, xp_2)
= addXPathToTGV (addConstraintToTGV (tgv, xp_1, false, c), xp_2, true)

A27 ifIH (createIfThenElseHyperlink (c,
getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_1),
getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_2),
getTGVNode (addIH (tgv, xp_1, c, s_1, s_2, xp_2), xp_2))) = c

A28 thenIH (createIfThenElseHyperlink (c,
getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_1),
getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_2),
getTGVNode (addIH (tgv, xp_1, c, s_1, s_2, xp_2), xp_2)))
= getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_1)

A29 elseIH (createIfThenElseHyperlink (c,
getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_1),
getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_2),
getTGVNode (addIH (tgv, xp_1, c, s_1, s_2, xp_2), xp_2)))
= getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_1)

A30 resultIH (createIfThenElseHyperlink (c,
getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_1),
getTreePattern (addIH (tgv, xp_1, c, s_1, s_2, xp_2), s_2),
getTGVNode (addIH (tgv, xp_1, c, s_1, s_2, xp_2), xp_2)))
= getTGVNode (addIH (tgv, xp_1, c, s_1, s_2, xp_2), xp_2)))

Nous avons déterminé le *Type Abstrait* TGV, qui nous permet de bâtir une représentation logique englobant une grande partie du langage *XQuery*. A partir de cette représentation, nous allons pouvoir appliquer des transformations pour modifier celle-ci et obtenir des *TGV* différents, voir équivalents, nous permettant de faciliter l'optimisation de l'évaluation. Mais il nous faut tout d'abord proposer une traduction exacte entre *XQuery* et les *TGV*.

3.4.6 Conclusion

Les *TGV* est donc un modèle de représentation de requête *XQuery*, il est formalisé grâce aux Types Abstraits de Données qui permettent de donner la structuration de chaque élément nécessaire. Nous nous sommes efforcé de vérifier la complétude de chaque type abstrait (suffisamment d'axiomes pour décrire les propriétés), sa consistance (pas de contradictions dans les axiomes).

Nous avons pu identifier les nœuds et les liens de nœuds composant les motifs d'arbre. Les différents motifs d'arbre qui peuvent être composés de contraintes et de liens de contraintes, et reliés entre eux par des hyperliens. Enfin, le tout est intégré dans un type TGV permettant de récupérer chacune de ces informations. Le tableau 3.10 récapitule chacun des types abstraits présentés dans cette section.

Comme nous avons pu le voir jusqu'ici, des exemples de requêtes *XQuery* étaient associés aux différentes représentations. Nous allons maintenant proposer une traduction du langage *XQuery* vers notre représentation sous forme de *TGV*.

Type Abstrait		Description
Node	N	Élément composant le motif d'arbre. Le document XML doit contenir le label correspondant.
NodeLink	NL	Association des nœuds composant le motif d'arbre. Le document XML doit respecter cette association.
TreePattern	TP	Ensemble de nœuds et de liens de nœuds donnant la structure nécessaire qu'un document XML doit respecter
SourceTreePattern	STP	Motif d'arbre s'appliquant à un document source.
IntermediateTreePattern	ITP	Motif d'arbre s'appliquant à un document intermédiaire.
ReturnTreePattern	RTP	Motif d'arbre de construction de résultat à partir des nœuds le composant.
AggregateTreePattern	ATP	Motif d'arbre de construction de résultat contenant une fonction d'agrégation.
Constraint	C	Type permettant l'héritage des types de données pour les contraintes.
Predicate	P	Prédicat entre un nœud et une chaîne de caractères.
ListParamater	LP	Ensemble des paramètres composant une fonction.
Function	F	Fonction applicable sur un élément des TGV, avec une liste de paramètres.
ConstraintLink	CL	Lien entre une contrainte et un élément des TGV. Il permet la composition des fonctions et des prédicats.
Hyperlink	HY	Lien permettant de relier les nœuds, les contraintes et les motifs d'arbre ensemble.
JoinHyperlink	JH	Hyperlien de jointure entre deux nœuds avec contrainte.
ConstraintHyperlink	CH	Hyperlien de contrainte composant un ensemble de contraintes et le reliant à un motif d'arbre de construction de résultat.
DirectionnalHyperlink	DH	Type d'hyperlien pour lesquelles un élément ne peut être ciblé que par un hyperlien directionnel unique.
ProjectionHyperlink	PH	Hyperlien de projection de valeur entre deux nœuds.
ExplorationHyperlink	EH	Hyperlien d'exploration à partir d'un nœud vers un motif d'arbre.
GeneralizedHyperlink	GH	Hyperlien de généralisation des résultats à partir d'un motif d'arbre vers un nœud.
SetHyperlink	EH	Hyperlien ensembliste reliant plusieurs motifs d'arbre vers un autre par l'intermédiaire d'une opération ensembliste.
IfThenElseHyperlink	IH	Hyperlien conditionnel reliant une contrainte (if) à deux ensembles de valeurs (then/else) qui peuvent être des nœuds ou des motifs d'arbre. Le tout est projeté sur un nœud.
TreeGraphView	TGV	Le type Tree Graph View relie l'ensemble des types abstraits de données qui composent le modèle.

Tab. 3.10 – Tableau récapitulatif des Types Abstraits de Données des TGV

3.5 XQuery vers TGV

Le modèle TGV permet de définir une représentation de requêtes pour *XQuery* sous forme de graphes. Nous avons formalisé chacun de ses éléments, il nous faut à présent définir un algorithme de transformation d'une requête canonique vers les TGV afin de définir le domaine de représentation. Nous allons donc utiliser la spécification de la grammaire d'une *XQuery* canonique et les spécifications des types abstraits.

Le langage de requête *XQuery* proposé dans la définition d'une *XQuery* canonique (definition 3.2.2.3) nous permet de reconnaître l'intégralité des requêtes *XQuery* non-typées. Celles-ci peuvent être représentées grâce aux TGVs. Il nous faut maintenant établir une traduction de *XQuery* vers les TGV.

3.5.1 XQuery

Pour chaque requête *XQuery*, nous commençons par l'algorithme 26. Il permet de créer une variable *tgv* contenant un TGV (ligne 1) auquel nous allons ajouter chacun de ses éléments. Nous pouvons voir que la variable *tgv* est définie en *global*, ce qui nous permettra de l'utiliser dans n'importe quelle méthode. Ensuite, nous ajoutons un *RTP* créé à la ligne 2. Le motif d'arbre de construction de résultat global (dit «racine») ne comportant pas de variable de définition est associé au mot clé '*root*', permettant ainsi de l'identifier dans le TGV. Ensuite, *RTP* est ajouté au TGV à la ligne 3. Il permettra de construire le document XML résultat de la requête *XQuery*.

Algorithme 26 initTGV

input : string query
output : TGV

1: global tgv := `createTGV` ()
2: rtp := `createRTP` ('root')
3: `addTreePattern` (tgv, rtp)
4: **if** *existSetOperator* (query) **then**
5: *initSetOperator* (query, 'root')
6: **else**
7: $FLWR$ (query, rtp)
8: **end if**
9: **return** tgv

Pour toute requête *XQuery*, nous avons la possibilité d'avoir une Expression *Expr* ou une composition d'expression avec des opérateurs ensemblistes. Cette dualité est représentée par le test de la ligne 4, dont la fonction *existSetOperator* permet de voir s'il existe un opérateur ensembliste dans la requête.

Si nous sommes dans le cas d'une opération ensembliste, nous invoquons la méthode *initSetOperator* (ligne 5) qui permet de créer un *ExplorationHyperlink* et initialiser les deux sous-requêtes (Algorithme 27).

Dans le cas d'une expression FLWR, nous devons l'analyser par la méthode *FLWOR* invoquée à la ligne 7 (algorithme 26).

3.5.2 SetOperator

Lorsqu'une expression contient une opérateur ensembliste, nous devons identifier les deux ensembles résultants de chacune des sous requêtes *XQuery*.

Algorithme 27 initSetOperator

input : string query, string tp

1: q_1 := *firstSet* (query)
2: q_2 := *secondSet* (query)
3: operator := *setOperator* (query)
4: atp_1 := *createUniqueVariable* (tgv)
5: atp := `createATP` (atp_1, 'return')
6: `addTreePattern` (tgv, atp_1)
7: **if** *existSetOperator* (q_1) **then**
8: *initSetOperator* (q_1, atp_1)
9: **else**
10: *FLWR* (q_1, atp_1)
11: **end if**
12: *//répéter ligne 4 à 11 avec la requête q_2*
13: root := `addLast` (`createXPath` (),
`createElement` (`getVariable` (tp), true))
14: root := `addLast` (root, `createElement` (`getRoot` (tp), true))
15: `addEH` (tgv, atp_1, atp_2, root, tp)

Avant d'analyser les deux sous-requêtes, nous devons les identifier à l'aide des méthodes *firstSet* et *secondSet* par leur nouvelle variable respective q_1 et q_2 (ligne 1 et 2). Puis définir l'opérateur ensembliste (ligne 3).

Pour chacune des deux variables q_1 et q_2, nous devons initialiser un *AggregateTreePattern* pour représenter la clause de reconstruction de résultat de chacune d'elles. À cette fin, une nouvelle variable est créée (ligne 4), puis un *ATP* est défini avec la fonction 'return' pour agrégat (ligne 5) et nous l'ajoutons au *TGV* (ligne 6).

Les requêtes q_1 et q_2 peuvent elles-même contenir soit une opération ensembliste, soit une expression FLWR. Nous devons alors traiter les deux cas (ligne 7 et 11). Dans le cas d'une opération ensembliste (ligne 7), nous invoquons de nouveau la méthode *initSetOperator* avec la sous-requête et le *TreePattern* de projection atp_1. Dans l'autre cas, nous initialisons une expression FLWR (ligne 10) dont la clause *return* se projettera sur l'*ATP*, en effet, comme le *Type Abstrait ATP* hérite du *RTP*, nous pouvons les associer à celui-ci. Ces opérations s'appliquent de la même manière pour q_1 et q_2 (ligne 12).

Pour finir, nous devons relier les deux *AggregateTreePatterns* avec l'opérateur ensembliste. Nous utilisons l'opération définie dans le *Type Abstrait* 25 : addEH. Cette opération permet de créer une opération ensembliste entre deux *TreePatterns*. Nous avons besoin pour cela d'un *XPath* de projection (ligne 13 et 14), celui-ci doit contenir la variable de déclaration en premier élément (ligne 13) et la racine en second (ligne 14). Enfin, nous pouvons appeler l'opération *addEH* avec les deux *ATP*s qui se projettent, l'*XPath* de projection, et l'opération ensembliste (ligne 15).

3.5.3 FLWR

Dans la plupart des cas, une requête *XQuery* canonique non typée s'exprime sous la forme d'une requête FLWR.

Algorithme 28 FLWR

input string query, RTP rtp

1: **if** *exists* (query, 'for') **then**
2: *initFor* (query)
3: **end if**
4: **if** *exists* (query, 'let') **then**
5: *initLet* (query)
6: **end if**
7: **if** *exists* (query, 'where') **then**
8: *initWhere* (query)
9: **end if**
10: *initReturn* (query, rtp, `addLast` (`createXPath` (), `createElement` (`getVariable` (rtp), true)))

L'algorithme 28 montre la décomposition des clauses d'une expression FLWR :
- **for** (ligne 1 et 2) : avec la fonction *initFor* ;
- **let** (ligne 3 et 4) : avec la fonction *initLet* ;
- **where** (ligne 5 et 6) : avec la fonction *initWhere* ;
- **return** (ligne 7) : qui est obligatoire et est initialisée par la fonction *initReturn*, avec comme *XPath* racine équivalent au nom du *RTP*.

3.5.4 For

La clause *for* d'une requête *XQuery* contient une variable de déclaration et un domaine de définition. Ce domaine est défini de deux manières, soit avec un ensemble de collection et un *XPath*, soit avec une redéfinition de domaine avec un *XPath* sur une variable définie préalablement. Ce qui s'exprime dans le tableau ci-dessous :

Range := XPath
| Path
| "collection" "(" StringLitteral (SetOperator StringLitteral)* ")"
(Path) ? ;

Un XPath doit contenir une variable prédéfinie, suivie par un Path. Si nous avons un Path en définition de domaine, cela correspond à la collection "*".

Ainsi, dans l'algorithme 29, chaque variable déclarée dans la clause "for" sera stockée dans la variable globale *fors* (ligne 1) qui nous permettra d'identifier si un *XPath* est optionnel ou obligatoire. En effet, si un *XPath* est défini au même niveau que la déclaration de sa variable, il est alors obligatoire. Dans le cas où l'*XPath* est utilisé dans une clause "let" et la variable définie dans une clause "for" externe, l'*XPath* sera optionnel car imbriqué.

Algorithme 29 initFor

input : string query

```
1: global fors := list ()
2: for chaque clause for f de query do
3:    var := getVariable (f)
4:    fors := add (fors, var)
5:    path := getXPath (f)
6:    if existSetOperator (f) then
7:       tp := createSTP (var, ", path)
8:       addTP (tgv, tp)
9:       initSetOperator (f, var)
10:   else if existCollection (f) then
11:      collection := getCollection (f)
12:      tp := createSTP (var, collection, path)
13:      addTP (tgv, tp)
14:   else
15:      b := exists (fors, first (path))
16:      addSH (tgv, path, var, b)
17:   end if
18: end for
```

Nous pouvons alors étudier chacune des clauses "for" f (ligne 2). Pour chacune d'entre elles, nous devons identifier sa variable (ligne 3), l'ajoutée à l'ensemble des variables définies dans la variable $fors$ (ligne 4) puis identifier l'*XPath* de définition de domaine (ligne 5). Enfin, nous pouvons identifier trois cas de définition de domaines : opération ensembliste, collection et redéfinition d'*XPath*. Les opérations ensemblistes génère un *SourceTreePattern* de projection *SourceTreePattern* de projection dont la racine vide (ligne 7). Il est ensuite rajouté au tgv (ligne 8) et enfin initialiser l'opération avec la méthode *initEnsemblistOperator* (ligne 9). Chaque collection est identifiée par son nom (ligne 11). Puis nous ajoutons un nouveau *SourceTreePattern* au tgv (ligne 12 et 13).

Dans le cas d'un *XPath* seul contenant une variable comme racine nous devons définir un *IntermediateTreePattern*. Tout d'abord, il est nécessaire de savoir si cet *XPath* doit être optionnel, c'est à dire de voir si la variable contenue dans le premier élément de l'*XPath* est définie dans la liste de variables "$fors$". Si la variable n'existe pas dans la liste "$fors$", le *SetHyperlink* sera alors optionnel. Pour créer un *ITP*, nous utilisons l'opération du TA TGV (TA 25) : *addSH* (ligne 16), qui initialise un *SetHyperlink* et un *ITP* avec la variable définie.

3.5.5 Let

La clause *let* nous permet de déclarer soit une sous-requête *XQuery* , soit une opération d'agrégat . La sous-requête est imbriquée dans la clause *return*, alors que l'opération d'agrégat s'applique à un *XPath*. Nous devons utiliser un *AggregateTreePattern* pour projeter l'ensemble de ces motifs d'arbre générés.

Algorithme 30 initLet

input : string query

1: **for** chaque clause let l de query **do**
2: var := $getVariable$ (l)
3: **if** $existFunction$ (f) **then**
4: function := $getFunction$ (f)
5: atp := `createATP` (var, function)
6: `addTP` (tgv, atp)
7: $path_1$:= $getXPath$ (f)
8: $path_2$:= `addLast` (`createXPath ()`, var)
9: b := $exists$ (fors, `first` ($path_1$))
10: tgv := `addPH` (tgv, $path_1$, $path_2$,b)
11: **else if** $existSetOperator$ (f) **then**
12: atp := `createATP` (var, 'return')
13: `addTP` (tgv, atp)
14: $initSetOperator$ (f, var)
15: **else**
16: atp := `createATP` (var, 'return')
17: `addTP` (tgv, atp)
18: $FLWOR$ (f, var)
19: **end if**
20: **end for**

Dans l'algorithme 30, nous pouvons voir que pour chaque clause "let" nous récupérons sa variable de définition (ligne 1 et 2). Ensuite, nous identifions les différents cas possibles de définition.

- Si c'est une fonction d'agrégat (ligne 3), nous pouvons ajouter un *AggregateTreePattern* avec la fonction et la variable associée (ligne 4 à 6). Ensuite, nous devons identifier l'*XPath* sur lequel nous appliquons cette opération d'agrégat (ligne 7). Cela définira un *ProjectionHyperlink* de cet *XPath* $path_1$ vers la racine de l'ATP $path_2$. L'optionalité de cet hyperlien est définie en fonction des variables définies dans les clauses "for" grâce à la liste "$fors$" (ligne 9). Enfin, nous relions les deux *XPath*s grâce à l'opération de création d'un *ProjectionHyperlink* (ligne 10) du TA TGV.
- Si c'est une sous-requête contenant des opérateurs ensemblistes (ligne 11), nous créons un ATP que nous ajoutons au TGV (ligne 12 et 13). Ensuite, nous initions cet ensemble qui se projettera sur ce nouvel ATP (ligne 14).
- Enfin, si c'est une expression FLWOR, nous ajoutons un ATP (ligne 16 et 17) et nous appelons la méthode *FLWOR* (ligne 18) avec la variable du let.

3.5.6 Where

La clause "where" doit contenir un ensemble de prédicats et d'appels de fonctions, reliés entre eux par des opérateurs booléens (and/or). Pour chacun de ces prédicats, nous devons associer la contrainte à son *XPath*. Nous devons aussi relier chacune de ces opérations entre elles par des *ConstraintHyperlink*s. L'algorithme 31 permet d'initialiser l'ensemble des contraintes reliées entre elles par des hyperliens de

contraintes. La fonction *initConstraintHyperlink* est appelée pour relier la première contrainte à une contrainte toujours vrai (*createConstraint("true")*). Comme les hyperliens de contraintes prennent toujours deux éléments, il faut initialiser le premier avec une contrainte vrai (pour éviter le cas d'un prédicat unique). Une fois l'hyperlien de contrainte retourné par la fonction, celui-ci est associé au motif d'arbre de construction de résultat (*addConstraintHyperlinkToRTP*).

Algorithme 31 initWhere

input : string *where*, RTP *rtp*

1: ch := *initConstraintHyperlink* (*where*, `createConstraint ("true")`, true)
2: *rtp* := `addConstraintHyperlinkToRTP` (*rtp*, ch)

L'algorithme 32 permet de créer la hiérarchie de vérification des opérations booléennes de la clause *where*. Il faut donc créer un hyperlien de contrainte pour chaque opérateur booléen rencontré, et attacher les contraintes entre elles par cet intermédiaire. La fonction *existBooleanOperator* (ligne 1) permet de vérifier s'il existe un opérateur booléen dans la chaîne restante. Si oui, il faut identifier la partie avant l'opérateur (*first* (*where*)) et le groupe de prédicat après l'opérateur (*second* (*where*)). p_1 (ligne 2) est donc le prédicat précédent l'opérateur, et p_2 (ligne 3) est la chaîne correspondant au reste de la clause présente après l'opérateur booléen. Cet opérateur est initialisé par la fonction *booleanOperator* (ligne 4), la variable b est donc une valeur booléen *vrai* si l'opérateur est '*and*' et *faux* si l'opérateur est '*or*'. La hiérarchie des opérateurs booléens dépend de celui qui a été donné en entrée de la fonction : *operator*. Si celui-ci est vrai, il faut relier la contrainte p_1 à l'élément *e* donné en entrée. Puis la contrainte est initialisé (ligne 6) et l'hyperlien de contrainte (ligne 7). Le reste de la clause est initialisé (ligne 8) avec l'hyperlien de contrainte en entrée de la fonction ainsi que l'opérateur *b*. Dans le cas où la valeur d'entrée *operator* est faux, il faut relier le reste de la clause à p_1 avant de le relier à l'élément d'entrée *e*. Ainsi, la contrainte c_1 est créée (ligne 10), la fonction *initConstraintHyperlink* est appelée récursivement sur le reste de la clause (p_2) et relié à c_1 avec l'opérateur booléen *b* (ligne 11). L'hyperlien de contrainte est alors créé à la ligne 12 avec le nouvel hyperlien de contrainte (créé par récursivité) avec l'élément d'entrée *e*. Dans le cas où il n'existe pas d'opérateur booléen dans la clause *where* (ligne 14), il suffit de créer la contrainte *c* (ligne 15) et relier celle-ci avec l'élément d'entrée *e* par un hyperlien de contrainte (ligne 16). Il est retourné comme feuille dans les hyperliens de contraintes.

L'algorithme 33 permet d'initialiser une contrainte en fonction de la chaîne de caractère donnée en entrée, résultat de la décomposition de l'algorithme 32. Nous devons vérifier la nature de la contrainte : fonction, jointure et prédicat. Tout d'abord, quelque soit l'opération, nous récupérons un *XPath* auquel l'opération est jointe (ligne 1), ainsi que l'information sur son optionalité (ligne 2). Dans le cas d'une fonction (ligne 3), nous devons récupérer son nom (ligne 4), l'ensemble des paramètres qui lui sont associés (ligne 5 à 8) et créer la contrainte avec *createFunction* du TA 10 (ligne 9). La fonction est associée à l'*XPath* grâce à l'opération *addOperationToTGV* du TA TGV (25) (ligne 10). Dans le cas d'une jointure entre deux *XPath* (ligne 11), nous devons créer la contrainte de jointure (ligne 12 et 13), puis récupérer le second *XPath* (ligne 14) avec son optionalité (ligne 15), et créer un *JoinHyperlink* (ligne 16)

Algorithme 32 initConstraintHyperlink

input : string *where*, element *e*, boolean operator
output : ConstraintHyperlink

```
1: if existBooleanOperator (where) then
2:    p1 := first (where)
3:    p2 := second (where)
4:    b := booleanOperator (where)
5:    if operator then
6:       c1 := initConstraint (p1)
7:       ch := createConstraintHyperlink (e, c1, true)
8:       return initConstraintHyperlink (p2, ch, b)
9:    else
10:      c1 := initConstraint (p1)
11:      ch := initConstraintHyperlink (p2, c1, b)
12:      return createConstraintHyperlink (e, ch, false)
13:   end if
14: else
15:   c := initConstraint (where)
16:   return createConstraintHyperlink (e, c, operator)
17: end if
```

grâce à l'opération *addJH* du TA TGV (25). Enfin, pour le cas d'un prédicat (ligne 17), nous récupérons l'opérateur (ligne 18) et la valeur de comparaison (ligne 19), puis nous créons la contrainte (ligne 20) et l'ajoutons au tgv (ligne 21). Pour finir, la contrainte créée dans est dans tous les cas retournée (ligne 22) pour être utilisée dans la fonction récursive *initConstraintHyperlink*.

3.5.7 Return

La clause "return" contient différents cas possibles d'expressions :
- les *Balises* s'ajoutent à un *XPath* correspondant à un ensemble de *Node*s à ajouter au *RTP* pour chaque projection ;
- les *XPath* crééent des *ProjectionHyperlink*s entre le nouveau *Node* et le *RTP* ;
- les *If Then Else*, de la même manière que pour l'*XPath* nous devons ajouter un *XPath* et deux expressions dont leurs *ATP* se projettent sur le nouveau *Node* ;
- les nœuds *textuels* doivent être associés à l'*XPath* de balises pour ajouter un *Node* de texte.

Ainsi, l'algorithme de traduction d'une clause "return" génère un *ReturnTreePattern* qui sera représenté par un *RTP* dans le cas de la requête *XQuery*, et par un *ATP* pour les sous-requêtes (*let*).

Comme nous pouvons le voir dans l'algorithme 34, il faut créer un *XPath* correspondant à l'ensemble des balises qui nous permettra de construire un document XML en résultat. Cet *XPath* sera ajouté au *ReturnTreePattern* pour représenter l'arbre de construction (ligne 1 à 5). Ensuite, nous devons tester chacunes des possibilités pour les éléments contenus dans cette clause "return" (ligne 6). Si c'est une balise (ligne 7), nous devons la rajouter à l'*XPath* de construction de balises : *tags* (ligne

Algorithme 33 initConstraint

input : string where
output : constraint

1: path := *getXPath* (w)
2: b := *exists* (fors, `first` (path))
3: **if** *existFunction* (w) **then**
4: function := *getFunction* (w)
5: params := `createList` ()
6: **for** chaque paramètre p de w **do**
7: params := `addLast` (params, p)
8: **end for**
9: c := `createFunctionConstraint` (function, params)
10: tgv := `addConstraintToTGV` (tgv, path, b, c)
11: **else if** *existsJoin* (w) **then**
12: p = *getPredicate* (w)
13: c = `createConstraint` (p)
14: $path_1$:= *getSecondXPath* (w)
15: b_1 := *exists* (fors, `first` ($path_1$))
16: tgv := `addJH` (tgv, path, b, $path_1$, b_1, c)
17: **else**
18: p = *getPredicate* (w)
19: s = *getValue* (w)
20: c := `createPredicateConstraint` (p, s)
21: tgv := `addConstraintToTGV` (tgv, path, b, c)
22: **end if**
23: **return** c

8). La variable *rest* (ligne 9) correspond au texte compris entre la balise ouvrante i et la fermante du même nom. Ce texte sera de nouveau analysé par la fonction *initReturn* (ligne 10). Si c'est un texte (ligne 11), Nous ajoutons un nœud feuille dans le *RTP*. Ce nœud de textuel est représenté par des doubles quotes ' " ' (ligne 12). Pour un *XPath* (ligne 13), nous récupérons un TA *XPath* (ligne 14), puis nous créons un *XPath* de projection (ligne 15). Pour une variable d'un *ATP*, nous devons alors créer un hyperlien de généralisation (ligne 16) puisqu'il faut généraliser l'ensemble résultat de l'*ATP*. Sinon, nous ajoutons un *ProjectionHyperlink* (ligne 19) et son *XPath* associé. Enfin, dans le dernier cas, nous avons un hyperlien conditionnel (*IfThenElseHyperlink*). Il faut identifier l'*XPath* (ligne 22), la contrainte (ligne 23), les deux sous-requêtes (ligne 24 et 25). Ensuite, nous créons l'*XPath* de projection (ligne 26) et initialisons l'ajout de l'*IfThenElseHyperlink* (ligne 27).

3.5.8 Conclusion

L'algorithme de traduction d'une requête *XQuery* canonique vers les TGV se décompose donc en plusieurs algorithmes prenant chaque caractéristique du langage pour en faire un algorithme. Ainsi, chaque mot clé est traduit naturellement entre *XQuery* et le modèle. La traduction est la dernier processus de transformation d'une requête *XQuery* vers notre modèle.

Algorithme 34 initReturn

input : string return, RTP rtp, XP xp

```
1: tags := createXPath ()
2: while !isEmpty (xp) do
3:    tags := addLast (tags, first (xp))
4:    xp := deleteFirst (xp)
5: end while
6: for chaque item i de return do
7:    if isTag (i) then
8:       tags := addLast (tags, createElement (i, true))
9:       rest := removeTag (i)
10:      initReturn (rest, rtp, tags)
11:   else if isText (i) then
12:      tgv := addXPathToTGV (tgv, addLast (tags, createElement ("i", true), true), true)
13:   else if isXPath (i) then
14:      xp := getXPath (i)
15:      projection := addLast (tags, createElement ("", true))
16:      if isEmpty (deleteFirst (xp)) then
17:         tgv := addGH (tgv, xp, projection)
18:      else
19:         tgv := addPH (tgv, xp, projection, false)
20:      end if
21:   else
22:      xp := getXPath (i)
23:      c := createConstraint (getConstraint (i))
24:      query1 := first (i)
25:      query2 := second (i)
26:      projection := addLast (tags, createElement ("", true))
27:      tgv := addIH (tgv, xp, o, query1, query2, projection)
28:   end if
29: end for
```

3.6 Conclusion

Nous avons présenté dans ce chapitre un modèle de représentation de requête complet pour XQuery : les *TGV*. Chaque requête est d'abord canonisée pour donner une structure adéquate pour les TGV. La requête canonique est alors traduite dans notre modèle de représentation où chaque caractéristique du langage est transformé en un élément des TGV.

L'étape de canonisation des requêtes XQuery s'appuie sur les travaux de [Chen 2004]. Nous avons étendu ses règles pour pouvoir transformer d'autres caractéristiques du langage (section 3.2). Ainsi, les ordonnancements, les opérations ensemblistes, les opérations conditionnelles, les séquences et les fonctions sont transformés en une forme équivalente que notre modèle est capable de reconnaître.

Les *TGV* sont un modèle de représentation de requête complet pour XQuery. Définis (section 3.3) et formalisés à l'aide des Types Abstraits de Données (ADT) (section 3.4), ils permettent de garder chacune des informations d'une requête XQuery. Une

représentation graphique est définie pour les TGV, permettant d'avoir une représentation intuitive des requêtes XQuery.

Afin d'automatiser la traduction d'une requête XQuery canonique vers le modèle de représentation, un algorithme complet de traduction est proposé dans la section 3.5. Cet algorithme est décomposé en plusieurs algorithmes décomposant chaque caractéristique du langage XQuery. Pour finir, un exemple de canonisation et de représentation de requête XQuery est détaillé dans son ensemble dans la section 3.1.

Nous obtenons donc un modèle de représentation complet pour le langage de requêtes XQuery. TGV peut aussi modéliser d'autres langages (SQL ou OQL) même s'il a été spécifiquement pensé pour XQuery. Cette représentation est intuitive, permettant d'identifier les documents ciblés par la requête, ainsi que leurs filtres associés. Grâce à notre algorithme de traduction du langage XQuery vers les TGV, nous pouvons vérifier la validité de notre modèle de représentation par l'intermédiaire des cas d'utilisation (*use-cases*) du [W3C 2006b]. Le chapitre 5 donne le détail de chacun de ces cas d'utilisation. Le cas d'utilisation *XMP* est détaillé en annexe F dans lequel est représenté chaque TGV correspondant aux requêtes données.

Le modèle de représentation TGV est conçu pour faciliter les manipulations. Des règles de transformation permettent de transformer un TGV pour améliorer son évaluation. Ainsi dans le cadre de l'optimisation que nous étudions dans le chapitre 4, nous développons l'évaluation des TGV, un langage de définition de règles de transformation et une stratégie de recherche. De plus, les relations entre les composants et l'isolation des domaines caractéristiques mis en évidence par les TGV permettent d'intégrer une annotation qui facilite l'intégration du modèle de coût. Cette approche est développée dans le chapitre suivant.

Chapitre 4

Optimisation des TGV

Comme nous avons pu le voir au chapitre 3, les *TGV*s nous permettent de modéliser le langage de requêtes XQuery grâce à un ensemble de définitions riches étendant les principes des *Tree Patterns*. Nous pouvons traduire une requête *XQuery* en un modèle de représentation graphique complet.

Dans ce chapitre, nous étudions l'optimisation de requêtes basée sur le modèle TGV. Pour cela, il est nécessaire de définir une algèbre d'évaluation correspondant à notre représentation de requêtes, un modèle de coût, des règles de transformation et une stratégie de génération de TGV équivalents pour nous permettre d'obtenir une représentation optimale. Les TGV permettent de représenter les requêtes *XQuery* en traduisant chaque caractéristique du langage. Pour évaluer les requêtes à partir d'uun TGV, il est nécessaire de définir une algèbre composée d'un ensemble d'opérateurs combinables en expressions résultant de la traduction des TGV. La définition de cette algèbre ressemble à la correspondance, dans le modèle relationnel, entre l'algèbre relationnelle (algèbre abstraite que nous allons introduire) et les arbres algébriques (correspondant aux TGV).

L'optimisation de requêtes repose sur une comparaison de mesures de qualité entre plans d'exécution. La mesure d'un plan se nomme le *coût d'évaluation*, il est donné par le *modèle de coût* permettant de donner une mesure à chacune des opérations composant le plan. Ainsi, entre deux plans, celui dont le coût d'évaluation est inférieure à l'autre est plus optimal. Le but de l'optimiseur est donc de trouver le plan le plus optimale ou au moins un proche de l'optimal.

Donner un coût d'évaluation à un *TGV* annoté représentant un plan d'exécution dans notre contexte de médiation demande un ensemble de formules de calculs constituant le modèle de coût. Un ensemble d'informations extérieures aux TGV doit être intégré pour qu'un coût soit calculable. Une base d'annotation est nécessaire pour associer les informations de coût au modèle de représentation (TGV). Plus précisément, le modèle de coût s'appuie sur les annotations pour calculer le coût d'un TGV annoté. La base d'annotation doit pouvoir intégrer toute sorte d'information à un

niveau de granularité quelconque dépendant des besoins du modèle de coût.

Les règles de transformations permettent de transformer les TGV. Ces transformations modifient le coût et visent à optimiser l'évaluation. L'ensemble des règles définies dans un optimiseur peut varier en fonction du contexte et aussi varier au cours du temps. Il est donc nécessaire de définir un système d'intégration de règles de transformation dans l'optimiseur proposé pour le rendre extensible. Un langage de définition de règles sera donc nécessaire pour enrichir l'optimiseur, celui-ci devient alors un *optimiseur extensible* par addition de règles de transformation.

Enfin, le choix des règles de transformation n'est pas arbitraire. Il faut définir un ordre d'application des transformations pour éviter de parcourir l'espace de recherche complet (l'ensemble de toutes les représentations possibles). A cette fin, une stratégie de recherche est nécessaire pour orienter le choix des règles de transformation afin que l'optimisation donne le plus rapidement possible un ensemble plus restreint de plans proches de l'optimal. Ainsi, nous pouvons éviter que le processus d'optimisation de la requête soit trop lent en limitant les parcours de l'espace de recherche.

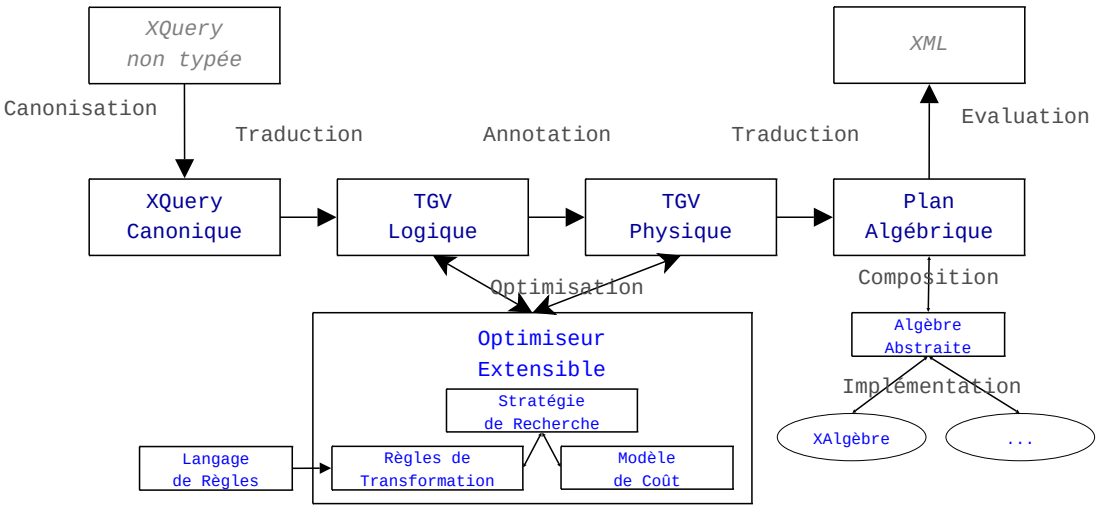


FIG. 4.1 – Cycle de traitement d'une requête XQuery

Le diagramme de la figure 4.1 montre les transformations successives effectuées sur un TGV pour permettre son évaluation. Les trois premières étapes (XQuery non typée, XQuery canonique, TGV logique) ont été présentées dans le chapitre précédent. La transition entre les deux étapes, *TGV logique* et *TGV physique*, se fait par l'annotation des *TGV*. L'optimiseur extensible interagit avec ces deux étapes grâce à la stratégie de recherche qui va choisir les règles de transformation adéquates, tout en s'appuyant sur le modèle de coût. Les règles de transformation sont définies à l'aide d'un langage de règles ; elles peuvent enrichir l'optimiseur extensible. Une fois l'étape d'optimisation effectuée, le TGV physique est alors traduit en plan algébrique. Ce plan algébrique dépend de l'algèbre qui est défini dans le système d'évaluation. Pour faciliter l'implantation des TGV dans les SGBD, nous proposons une algèbre abstraite définissant les primitives des opérations à opérer sur les TGV. Ainsi, il suffit

de définir, lors de la conception du système, une traduction de l'algèbre abstraite vers l'algèbre implémentée pour évaluer le TGV. Enfin, la requête est évaluée pour donner un ensemble de fragments XML en résultats.

Nous étudions dans ce chapitre une algèbre logique pour l'évaluation de TGV (section 4.1). Pour nous permettre d'insérer de nouvelles informations sur les TGV, nous proposons un modèle d'annotation, notamment pour introduire un modèle de coût pour les TGV (section 4.2). Puis nous introduisons le langage de définition de règles permettant de définir l'extensibilité de l'optimiseur (section 4.3). Pour finir, nous proposons une stratégie de recherche pour manipuler les transformations et orienter l'optimiseur (section 4.4).

4.1 Algèbre Abstraite pour TGV

Pour évaluer les requêtes *XQuery*, il est nécessaire de définir une algèbre d'évaluation. Ainsi, des expressions d'opérateurs algébriques définiront la manière de procéder par opérations successives ou parallèles pour produire un résultat de requête en XML. Comme chaque requête est représentée sous forme d'un *TGV*, il est nécessaire que l'arbre algébrique soit dérivé de cette représentation des requêtes. Nous proposons tout d'abord une algèbre abstraite, indépendante de toute considération physique, ceci pour traduire un TGV en modèle d'évaluation logique. Le but de cette algèbre abstraite est de définir les primitives des opérations de traitements sur les TGV. Ainsi, cette algèbre abstraite permet de faciliter l'implantation des TGV dans un système grâce à une traduction de ces opérations vers les opérations du système considéré.

L'algèbre abstraite est donc un ensemble d'opérations de traitements définies pour les TGV. Chaque opération est une primitive qui peut ensuite être étendue dans l'algèbre physique considérée. Dans notre système de médiation *XLive* présenté dans le chapitre 5, nous avons choisi d'effectuer la traduction de l'algèbre abstraite vers la *XAlgèbre*.

L'algèbre abstraite pour l'évaluation de TGV doit permettre de traiter des documents XML grâce aux éléments définis dans le chapitre 3 : nœuds, motifs d'arbre, contraintes, hyperliens. Ainsi, chaque type d'élément correspond à une opération particulière qui peut produire un nouvel ensemble de documents XML.

Les opérateurs de l'algèbre utilisent les informations contenues dans les motifs d'arbre pour transformer des fragments de documents XML. Les motifs d'arbre d'un TGV donnent les informations nécessaires pour définir les transformations d'arbres XML :
- le filtrage des éléments est donné par le motif des motifs d'arbre source et intermédiaire ;
- la restriction des arbres est donnée par les contraintes reliées entre elles par des hyperliens de contraintes ;
- les fonctions d'agrégats sont définies dans les motifs d'arbre d'agrégation ;

– la construction de documents XML est définie par les motifs d'arbre résultat ;
– les projections d'éléments sont données par les hyperliens directionnels, reliés à un motif d'arbre ;
– les jointures de fragments XML sont définies par les hyperliens de jointures.

Ces informations servent donc à définir l'algèbre abstraite. Chacune d'elles peut être retrouvée à partir d'un motif d'arbre auquel il est associé ; les contraintes et les jointures sont reliées à un motif d'arbre résultat, et les hyperliens directionnels sont liés au motif d'arbre de destination.

Un opérateur de l'algèbre prend en paramètre un ou plusieurs motifs d'arbre pour définir l'opération à effectuer. Il transforme un ensemble de fragments XML symbolisés par «τ», en un nouvel ensemble de fragments.

Les fragments de document XML produis par les opérateurs de l'algèbre ont pour racine le nom de variable de l'opérateur qui la créé. Cette racine permet d'identifier la provenance des fragments, utile dans le cas d'union de fragments de documents (par exemple, la jointure doit pouvoir identifier les deux racines distinctes).

L'opérateur d'évaluation des motifs d'arbre d'un TGV est défini par la fonction *eval*, qui prend comme ensemble de départ l'ensemble des motifs d'arbre le composant : «*tgv*». Grâce aux propriétés des motifs d'arbre, des règles de décomposition sont définies pour décomposer l'opérateur *eval* en opérateurs spécifiques de l'algèbre. Cette décomposition permet de produire une expression algébrique définissant de manière précise l'évaluation d'un TGV.

Par la suite, suivant leur type, les motifs d'arbres sont désignés par leur acronyme respectif provenant du modèle : *tp* pour un motif d'arbre quelconque, *stp* pour un motif d'arbre source, *itp* pour un motif d'arbre intermédiaire, *rtp* pour un motif d'arbre résultat et *atp* pour un motif d'arbre d'agrégation. De plus, comme nous travaillons sur des groupements de motifs d'arbres (motifs d'arbre reliés entre eux par des hyperliens), nous utilisons la valeur *tgv* pour représenter l'intégralité des motifs d'arbre présents dans une instance de TGV. Le symbole «E» représentera un sous-ensemble de motifs d'arbre. Ainsi, l'expression $tgv - rtp$ exprime l'ensemble des motifs d'arbres du *tgv* privé du motif d'arbre résultat *rtp*. Les définitions suivantes permettent de définir l'évaluation d'un TGV.

4.1.1 Définition des opérateurs

Les opérateurs de l'algèbre abstraite permettent de définir une évaluation d'un ensemble de documents XML à l'aide d'un groupement de motifs d'arbre. Chaque motif d'arbre possède des caractéristiques provenant du langage XQuery, certaines de celles-ci peuvent être traduites par des contraintes ou des hyperliens liés aux motifs d'arbre. Ainsi, pour l'algèbre abstraite, six opérateurs sont définis pour le traitement d'ensemble de fragments XML. Chaque opérateur prend en paramètre un groupement de motifs d'arbre *tp* et s'applique sur un ensemble de fragments XML τ. Le résultat d'un opérateur produit un nouvel ensemble de documents. À

$$eval_{\{tp_1 \ \cup \ tp_2\}}(\tau) = eval_{tp_1}(\tau) \cup eval_{tp_2}(\tau)$$

TAB. 4.1 – Distributivité de l'opérateur *eval* sur des motifs d'arbre disjoints

chaque opérateur est associé en indice son paramètre (noms de variable des motifs d'arbre) et entre parenthèses l'ensemble de fragments XML τ : $f_{tp}(\tau)$.

Nous pouvons définir l'évaluation d'un TGV à l'aide des opérateurs suivants :
- *eval* : Opérateur d'**évaluation** d'un ensemble de documents XML grâce à un groupement de motifs d'arbre en paramètre, elle est décomposable et distributive ;
- σ : Opérateur de **restriction** de l'ensemble de document XML. La restriction sélectionne les documents satisfaisant un ensemble de contraintes liés à un motif d'arbre résultat ;
- π : Opérateur de **projection** des valeurs d'un document XML, les valeurs sont choisies grâce aux hyperliens directionnels donnés par un RTP ;
- β : Opérateur de **construction** d'un document XML. Chaque nœud génère une balise, les fragments de τ sont utilisés pour introduire les valeurs dans ce nouveau document XML ;
- α : Opérateur d'**agrégation** sur un ensemble de documents XML τ en utilisant la fonction d'agrégat contenue dans un motif d'arbre d'agrégation ;
- ϕ : Opérateur de **filtrage** de documents XML grâce aux motifs d'arbres.

Définition 4.1 : Opérateur $eval_{tp}(\tau) \rightarrow \tau'$

L'opérateur *eval* évalue un ensemble de fragments XML grâce aux information provenant du groupement de motifs d'arbre qui le compose. les motifs d'arbre qu'il contient peuvent être décomposés en opérateurs de l'algèbre. La fonction *eval* est distributive sur les motifs d'arbre et les documents XML.

L'opération *eval* permet d'évaluer un ensemble de documents XML grâce à aux motifs d'arbres donnés en paramètre ; il produit en résultat un nouvel ensemble de document. Chaque ensemble de motifs d'arbre peut être décomposé en opérateur(s) en fonction des propriétés des groupements de motifs d'arbre qui le compose. La fonction *eval* est distributive par rapport à l'union des motifs d'arbre. La composition de l'opérateur *eval* se comporte à la manière d'une multiplication sur une addition. Ainsi, l'évaluation de fragments XML par l'union de motifs d'arbres disjoints (aucune opération commune entre les deux motifs d'arbre) équivaut à l'union des évaluations distinctes de fragments XML par chaque groupement de motif d'arbre (cf tableau 4.1).

Définition 4.2 : Opérateur $\sigma_{rtp}(\tau) \rightarrow \tau'$

L'opérateur σ restreint un ensemble de fragments τ à l'aide de contraintes. Les contraintes proviennent des hyperliens de contraintes reliées au motif d'arbre résultat *rtp*. τ' ne contient que les arbres de τ vérifiant cette composition de contraintes.

L'opérateur σ permet de restreindre l'ensemble de document τ à l'aide des contraintes associées au motif d'arbre résultat *rtp* par l'intermédiaire des hyperliens de contraintes (par héritage, un motif d'arbre d'agrégation *atp* possède des hyperliens de contraintes). Si la liste des contraintes reliée par les opérateurs booléens est vérifiée pour un arbre $t \epsilon \tau$, alors t est ajouté à τ'.

Définition 4.3 : Opérateur $\pi_{rtp}(\tau) \rightarrow \tau'$

L'opérateur π projette les valeurs de τ grâce aux hyperliens directionnels liés à *rtp*. τ' contient un ensemble de groupes d'éléments projetés, dont la racine est le nom de variable de *rtp*.

L'opérateur π permet de projeter des fragments de τ, à l'aide des hyperliens directionnels reliés aux éléments du motif d'arbre de *rtp*. Ainsi, un hyperlien directionnel projette les valeurs de τ dans un document XML dont la racine est le nom de variable de *rtp*. Lorsqu'un hyperlien directionnel est optionnel, un élément vide est produit dans le cas où la valeur requise n'apparaît pas dans τ.

Définition 4.4 : Opérateur $\beta_{rtp}(\tau) \rightarrow \tau'$

L'opérateur β construit des documents XML grâce au motif d'arbre contenu dans *rtp*. Pour chaque fragment XML de τ, les nœuds du motif d'arbre produisent la structure du document XML. Les nœuds liés aux hyperliens directionnels prennent les valeurs correspondantes de τ pour les associer au nouveau document XML.

L'opérateur β permet de construire un nouveau document XML pour chaque fragment XML de τ. Chaque nœud du motif d'arbre *rtp* produit un élément du nouveau document XML, l'arborescence du motif d'arbre produit la structure du document. Si un nœud est relié à un hyperlien directionnel, il prend alors la valeur correspondante de τ et la rattache au document XML. Les nœuds dont le lien (*NodeLink*) est optionnel produisent un élément textuel. La racine de chaque document XML a pour nom la variable de *rtp*.

Définition 4.5 : Opérateur $\alpha_{atp}(\tau) \rightarrow \tau'$

L'opérateur α applique une fonction d'agrégation sur l'ensemble τ. La fonction d'agrégation provient de la contrainte associé au motif d'arbre d'agrégation *atp*. chaque fragment généré est associé à la variable de *atp*.

L'opérateur α permet d'agréger des fragments de τ à partir de la fonction d'agrégat de *atp*. τ' est l'association de τ et du résultat de l'opérateur α associé à la variable de définition de *atp* (un agrégat est un *let*, il créé un nouvel ensemble, sans supprimer l'ancien). Dans le cas d'une requête imbriquée, la fonction d'agrégat groupe les valeurs requises par les contraintes liées au motif d'arbre (σ).

Définition 4.6 : Opérateur $\phi_{tp}(\tau) \rightarrow \tau'$

L'opérateur ϕ filtre les fragments de τ grâce au motif d'arbre *tp*. Chaque fragment de τ contenant le motif obligatoire de *tp* est ajouté à τ' associé au nom de variable de *tp*. Les fragments de τ' contiennent les éléments requis par le motif d'arbre, qu'ils soient obligatoires ou optionnels (s'ils sont présents).

L'opérateur ϕ permet de filtrer l'ensemble de document τ grâce au motif d'arbre de *tp*. Tout fragment τ ne contenant pas le motif obligatoire n'est pas ajouté à τ'. Les fragments de τ' contiennent le motif d'arbre requis, ainsi que les balises optionnelles si celles-ci sont présentes. La racine de chaque fragment est donné par le nom de variable de *tp*.

Les opérateurs de l'algèbre abstraite que nous venons de détailler définissent l'évaluation de motifs d'arbre sur un ensemble de documents XML. Chaque opérateur prend les caractéristiques de ces motifs d'arbre pour manipuler les fragments de document XML. Les motifs d'arbre proviennent d'un TGV décomposé. Les règles de décomposition de l'opérateur *eval* proposé dans la section 4.1.2 génère un arbre algébrique à partir des motifs d'arbre composant un TGV.

4.1.2 Algorithme de décomposition

La décomposition de l'opérateur *eval* de l'algèbre abstraite repose sur les propriétés des motifs d'arbre. Celle-ci commence par le motif d'arbre résultat du TGV (unique) qui ne possède pas d'hyperlien de projection sortant, donc indépendant. Le tableau 4.2 montre la décomposition récursive de cette fonction. Elle est composée de neuf règles de décomposition que nous détaillons par la suite. La décomposition débute par la règle n°1 (motif d'arbre résultat).

Règle	Décomposition	
1	$eval_{tgv}(\tau)$	$\Leftrightarrow eval_{rtp}(\sigma_{rtp}(\tau_r))$
2	$eval_{rtp}(\tau)$	$\Leftrightarrow \beta_{rtp}(\pi_{rtp}(\tau))$
3	$eval_{tgv-rtp}(\tau)$	$\Leftrightarrow eval_{E_1 \cup ... \cup E_n}(\tau)$ $\Leftrightarrow eval_{E_1}(\tau) \cup ... \cup eval_{E_n}(\tau)$
4	$eval_{E_x}(\tau)$	$\Leftrightarrow eval_{atp_x}(\sigma_{atp}(\tau_x))$
5	$eval_{atp}(\tau)$	$\Leftrightarrow eval_{rtp_a}(\alpha_{atp}(\tau))$
6	$eval_{E-atp}(\tau)$	$\Leftrightarrow eval_{E_1 \cup ... \cup E_n}(\tau)$ $\Leftrightarrow eval_{E_1}(\tau) \cup ... \cup eval_{E_n}(\tau)$
7	$eval_{stp}(\tau)$	$\Leftrightarrow \phi_{stp}(\tau)$
8	$eval_{itp_s}(\tau)$	$\Leftrightarrow eval_{itp}(eval_{tp_s}(\tau))$
9	$eval_{itp}(\tau)$	$\Leftrightarrow \phi_{itp}(\pi_{itp}(\tau))$

TAB. 4.2 – Règle de décomposition des motifs d'arbre

Nous détaillons maintenant les règles définies dans le tableau 4.2 :

Règle 1 **Décomposition avec un motif d'arbre résultat**. Le motif d'arbre résultat est retiré de l'ensemble des motifs d'arbres *tgv* pour produire un opérateur lui correspondant. L'opération σ utilise les hyperliens de contraintes provenant du *rtp* pour restreindre l'ensemble τ_r utilisé dans $eval_{tgv}(\tau)$. τ_r correspond à l'évaluation de de τ par le groupement des motifs d'arbre restant dans «$tgv - rtp$» : $\tau_r = eval_{tgv-rtp}(\tau)$. Ainsi les arbres de τ_r qui ne vérifient pas $\sigma_{rtp}(\tau_r)$ sont supprimés, les autres sont projetés dans l'opérateur π de la règle n°2 ;

Règle 2 **Projection des valeurs et construction de résultat**. A partir de la règle n°1, nous décomposons l'opérateur *eval* dont le paramètre est un motif d'arbre résultat (*rtp*). Ce motif d'arbre se décompose sous la forme d'un opérateur de projection (π) obtenu grâce aux hyperliens directionnels (liés à *rtp*) et d'un opérateur de construction de document (β) grâce au motifs d'arbre de *rtp* ;

Règle 3 **Décomposition avec** $tgv - rtp$. Grâce à la propriété de distributivité de l'opérateur *eval* sur les motifs d'arbre disjoints, le groupement $tgv - rtp$ peut être décomposé en ensembles disjoints de motifs d'arbre ($E_1 \cup ... \cup E_n$). Chaque groupe de motifs d'arbre E_x contient un ensemble de *stp*, *itp* et *atp* reliés entre eux par des hyperliens (non encore traités). Deux groupes de motifs d'arbre E_x et E_y sont dits disjoints s'il n'existe aucun hyperlien directionnel (non encore traités) les reliant. Ainsi, nous pouvons déduire grâce à la distributivité de l'opérateur *eval* sur les groupes E_x l'expression suivante : $eval_{E_1}(\tau) \cup ... \cup eval_{E_n}(\tau)$;

Règle 4 **Décomposition avec un groupe de motifs d'arbre contenant un motif d'arbre d'agrégation**. La décomposition de l'opérateur *eval* avec un groupe de motifs d'arbre E_x contenant un motif d'arbre d'agrégation est équivalente à la décomposition de la règle n°1. En effet, un motif d'arbre d'agrégat hérite des propriétés du motif d'arbre résultat. Ainsi, le sous-ensemble associé au motif d'arbre *atp* est donné par l'expression : $E_x - atp$. L'ensemble des contraintes liées au motif d'arbre d'agrégation par les hyperliens de contraintes restreint l'ensemble τ_x produit par l'évaluation de τ avec le groupe de motifs d'arbre $E_x - atp$.

Règle 5 **Décomposition avec un motif d'arbre d'agrégation**. La décomposition de l'opérateur *eval* avec un motif d'agrégation seul donne l'évaluation de l'opérateur α sur τ. Alors, α effectue un agrégat des valeurs de τ avec la fonction

d'agrégat contenue dans *atp*. De plus, grâce à l'héritage du motif d'arbre résultat, la décomposition génère l'opérateur $eval_{rtp_a}$ à partir du motif d'arbre résultat contenu dans *atp* (puis décomposé dans la règle n° 2).

Règle 6 **Décomposition avec $E - atp$.** Grâce à la propriété de distributivité de l'opérateur *eval* nous pouvons décomposer en ensembles distincts E_x disjoints (de la même manière que la règle n°3, mais avec un *atp*). Le groupe E_x correspond à un ensemble de motifs d'arbre reliés à un motif d'arbre d'agrégation (qui a été décomposé). Chaque groupe de motifs d'arbre disjoint E_x de E contient un nouveau sous ensemble motifs d'arbre liés entre eux et pouvant être décomposé. Ainsi, comme pour la règle n°3, l'opérateur *eval* avec le groupe de motifs d'arbre $E - atp$ donne l'expression algébrique : $eval_{E_1}(\tau) \cup ... \cup eval_{E_n}(\tau)$.

Règle 7 **Décomposition avec un motif d'arbre source**. L'opérateur *eval* avec un motif d'arbre source sur τ se décompose en un opérateur de filtrage ϕ sur τ. Le motif d'arbre source filtre les arbres présents dans l'ensemble τ pour produire un résultat contenant la hiérarchie des éléments définie par l'association des nœuds et des liens de nœuds contenus dans le motif d'arbre.

Règle 8 **Décomposition avec un motif d'arbre intermédiaire lié à un motif d'arbre**. L'opérateur *eval* avec un motif d'arbre intermédiaire (itp_s) lié à un autre motifs d'arbre se décompose en un opérateur de *eval* avec un motif d'arbre intermédiaire indépendant (*itp*). Il s'applique sur la décomposition de l'opérateur *eval* avec le motif d'arbre lié (tp_s). Cet opérateur est décomposé dans la règle n°9.

Règle 9 **Décomposition avec un motif d'arbre intermédiaire indépendant**. L'opérateur *eval* associé à un motif d'arbre intermédiaire indépendant se décompose en un opérateur de filtrage ϕ appliqué sur un opérateur de projection π. Le filtrage est donné par le motif d'arbre de *itp*, tandis que la projection des fragments de τ provient de l'hyperlien d'exploration de *itp*.

Les règles de décomposition permettent de définir la transformation de l'opérateur *eval* en fonction des motifs d'arbre qui lui sont associée. L'algorithme 35 donne la fonction de décomposition récursive d'un TGV à l'aide des règles de décomposition.

L'algorithme 35 détaille l'utilisation des règles de décomposition du tableau 4.2. Cette fonction récursive utilise un ensemble de motifs d'arbre *tps* sur un ensemble de document XML «τ». Dans cet algorithme, nous avons besoin de trois fonctions particulières : *existsGroups*, *exists* et *getTP*.

La fonction *exists* permet de savoir s'il existe un motif d'arbre "*racine*" (aucun hyperlien directionnel ne part de ce motif d'arbre) dont le type est équivalent au mot donné. La fonction *getTP* permet de retourner le motif d'arbre "*racine*" requit par le type donné. La fonction *existsGroups* permet de déterminer s'il existe des groupes de motifs d'arbre contenant une "*racine*".

L'algorithme détaille les décompositions en fonction du motif d'arbre "*racine*" déterminé par la fonction *exists*. Ainsi, la première condition (ligne 1 à 6) permet de voir s'il existe plusieurs ensembles de motifs d'arbre racine (ligne 1), alors le résultat retourné (ligne 6) est l'union des décompositions de chacun de ces ensembles (ligne

Algorithme 35 composeOperations (tps, τ)

1: **if** existsGroups (tps) == true **then**
2: r := ;
3: **for all** distinct group E from tps **do**
4: r := $r \cup composeOperations(E, \tau)$;
5: **end for**
6: **return** r ;
7: **else if** exists (tps, "rtp") **then**
8: rtp := getTP (tps, "rtp") ;
9: r := composeOperations (tgv - rtp, τ) ;
10: **return** $\beta_{rtp}(\pi_{rtp}(\sigma_{rtp}(r)))$;
11: **else if** exists (tps, "atp") **then**
12: atp := getTP (tps, "atp") ;
13: r := composeOperations (tps - atp, τ) ;
14: a := $\alpha_{atp}(\sigma_{atp}(r))$;
15: **return** $\beta_{atp}(\pi_{atp}(a))$;
16: **else if** exists (tps, "itp") **then**
17: itp := getTP (tps, "itp") ;
18: r := composeOperations (tps - itp, τ) ;
19: **return** $\phi_{itp}(\pi_{itp}(r)$;
20: **else if** exists (tps, "stp") **then**
21: stp := getTP (tps, "stp") ;
22: **return** $\phi_{stp}(\tau)$;
23: **end if**

3 et 4). La seconde condition (ligne 7) correspond au motif d'arbre résultat, la décomposition de la règle n°1 est appliquée (ligne 9) et donne le résultat r, puis la règle n°2 (ligne 10) est appliquée à r. La troisième condition décompose les motifs d'arbre d'agrégation avec la règle n°4 (ligne 13) et donne le résultat r, la règle n°5 est appliquée (ligne 14) puis retournée. Puis le motif d'arbre intermédiaire est décomposé grâce à la règle n°8 (ligne 18), puis transformée à l'aide de la règle n°9 (ligne 19). Pour finir, les motifs d'arbre source sont transformés par la règle n°7 (ligne 22).

Ainsi, nous avons défini une méthode récursive permettant de générer un arbre algébrique. Cet algorithme utilise les règles de décomposition de l'opérateur *eval*. L'entrée de cet algorithme est un ensemble de motifs d'arbre, provenant à l'origine d'une TGV. Cette expression algébrique est utile pour définir le plan physique et son coût d'évaluation par opérateur. De plus, les règles de transformation équivalentes (section 4.3.2.1) doivent utiliser cette évaluation pour vérifier l'équivalence des résultats.

4.1.3 Exemple d'arbre algébrique

Afin d'illustrer une expression algébrique définie à partir d'un TGV, nous reprenons l'exemple de la figure 3.2 présenté à la page 39. Par application de l'algorithme (et des règles de décomposition), nous obtenons l'expression algébrique suivante :

1 :	$\beta_{rtp}(\pi_{rtp}(\sigma_{rtp}($
2 :	$\phi_{\$y}(\pi_{\$y}($
3 :	$\phi_{\$x}(\pi_{\$x}($
4 :	$\beta_{\$u_3}(\pi_{\$u_3}(\alpha_{\$u_3}(\sigma_{\$u_3}($
5 :	$\phi_{\$u_1}(\tau) \cup \phi_{\$u_2}(\tau)))))))))$
6 :	$\cup$
7 :	$\beta_{\$l_1}(\pi_{\$l_1}(\alpha_{\$l_1}(\sigma_{\$l_1}($
8 :	$\phi_{\$z}(\tau)))))))$

TAB. 4.3 – Formule algébrique du TGV de la figure 3.2

Le tableau 4.3 donne l'expression algébrique correspondant au TGV de la figure 3.2. L'algorithme commence par le motif d'arbre résultat (rtp), qu'il décompose à la ligne 1 avec les contraintes ($\$y//price > 5$ et $contains(\$x, "dauphin")$) associées à l'opérateur σ. Ensuite, les deux groupes de motifs d'arbre sont décomposés grâce à l'opérateur d'union à la ligne 6 entre les deux formules (ligne 2 à 5 et ligne 7 et 8). Le premier ensemble se concentre sur le motif d'arbre intermédiaire relié à l'union des deux motifs d'arbre source ($\$u_1$ et $\$u_2$). Ainsi, à la ligne 2 l'exploration du motif d'arbre \$y est donné par les opérateurs $\phi_{\$y}$ et $\pi_{\$y}$. De la même manière, le second motif d'arbre intermédiaire \$x est décomposé à la ligne 3, projetant les informations de l'agrégat défini dans le motif d'arbre $\$u_3$. Il est lui-même décomposé à la ligne 4. L'opérateur ensembliste d'union regroupe les deux motifs d'arbre source $\$u_1$ et $\$u_2$ dans l'opérateur $\sigma_{\$u_3}$. Celui-ci s'applique donc sur l'union des deux évaluations donné à la ligne 5.
Le deuxième ensemble de motif d'arbre (ligne 6) débute avec la décomposition du motif d'arbre d'agrégation $\$l_1$ (ligne 7), contenant le prédicat de jointure entre les paths \$y/@isbn et \$z/@isbn (les racines produites par les opérateurs donne le nom des variables à utiliser pour la jointure). La décomposition du motif d'arbre source \$z est défini à la ligne 8.

L'expression algébrique définie dans le tableau 4.3 donne l'évaluation d'un ensemble de documents provenant des collections «reviews» et «catalogs». Le mot «dauphin» doit apparaître dans le document, de plus, le prix du livre ne doit pas dépasser 15 euros. Cette expression algébrique contient les opérateurs de sélection pour les collections et l'opérateur d'union pour les deux collections. Ensuite, l'opérateur de restriction élimine les solutions ne correspondant pas à la requête. L'opérateur d'agrégation permet de construire les informations qui sont associées aux livres. Chaque élément nécessaire est projeté pour produire la valeur requise pour la reconstruction de résultat.

4.1.4 Conclusion

Nous avons défini une algèbre abstraite permettant de définir des expressions algébriques d'évaluation des TGV. Cette algèbre abstraite définie les primitives des opérations de traitement sur les TGV. Une simple traduction de ces opérations permet l'implantation des TGV dans les systèmes de gestion de base de données pour évaluer des documents XML. Un exemple de traduction est proposé dans le chapitre 5 avec la *XAlgèbre*.

Des opérateurs et un algorithme de décomposition sont proposés pour produire ces expressions algébriques à partir du modèle. Toutefois, cette représentation ne contient que des informations logiques[1] et il n'est pas possible de donner des informations physiques[2] pour définir de manière concrète les opérateurs de l'algèbre. À cette fin, un système d'annotation est ajouté aux TGVs pour associer des informations utiles pour leurs manipulations.

4.2 Annotation des TGV

Dans le chapitre précédent, la représentation TGV a été définie pour modéliser de façon logique une requête XQuery et faciliter sa représentation. Une algèbre abstraite a ensuite été associée à ce modèle de représentation pour permettre de définir son évaluation. Toutefois, des informations extérieures au modèle sont nécessaires pour permettre d'orienter l'optimisation. En effet, les TGVs ne suffisent pas pour intégrer des informations sur le modèle de coût permettant de prévoir le coût d'évaluation d'un TGV, des choix d'algorithmes physiques sur les opérations d'évaluation, des choix d'évaluation de motifs d'arbre locaux ou distants dans le cadre de la médiation. Ainsi, nous proposons un modèle d'annotation de TGV, qui permet de faciliter la modélisation du modèle de coût, les choix d'algorithme et de localisation, tout cela dans un but d'optimisation physique. Ainsi, nous parlons de *tgv physique* lorsque celui-ci est annoté et d'un *tgv logique* dans le cas où celui-ci ne possède pas d'annotations. Cette base d'annotation est utilisée dans le langage de règles (section 4.3.1) pour définir des transformations de tgv physiques.

Nous introduisons donc un modèle générique d'annotation permettant d'intégrer tout type d'annotation à notre modèle. Ainsi, il ne sera pas limité aux informations de coût, choix d'algorithme et de localisation, toute information pourra lui être associé via le modèle d'annotation.

4.2.1 Définition

Pour annoter un TGV, chaque information s'associe à un certain degré de granularité du modèle. Une information de coût peut aussi bien n'être associée qu'à un nœud, un lien de nœud, un ensemble de nœuds et de lien de nœuds, une contrainte, un motif d'arbre, un ensemble de motifs d'arbre, un hyperlien, ou qu'à tout autre composition des éléments du modèle. Ainsi, le modèle d'annotation doit pouvoir tenir compte de la granularité sur un ensemble d'éléments du tgv auquel sera associé l'annotation.

[1] Les informations provenant de la requête, sans prendre en compte le système.
[2] Les informations provenant du système que l'on peut associer à la requête.

Pour annoter les TGV, nous avons besoin de quatre types différents :
- **Information** : Information d'annotation ;
- **ListTGVObject** : Ensemble d'éléments composant un TGV (Node, NodeLink, Constraint, TreePattern, Hyperlink) ;
- **Annotation** : Lien entre une information à annoter et un ensemble d'éléments du TGV (annotation du TGV) ;
- **AnnotationSet** : Liste des annotations d'un TGV.

4.2.1.1 Information

L'information est un champ externe au TGV (coût, localisation, algorithme...) que l'on souhaite annoter. Cette annotation peut être modifiée par héritage, et ainsi des renseignements supplémentaires peuvent y être associé (formule, identifiant, valeur). Cet héritage permet de rendre spécifique chaque annotation.

Définition 4.7 : Information

Une *information* contient les renseignements relatifs à l'annotation. Il peut être étendu pour ajouté des opérations spécifiques.

4.2.1.2 Liste d'éléments

Chaque information est liée à un ensemble d'éléments, donnant la granularité de l'annotation. Nous définissons donc une liste d'éléments, permettant de garder les éléments liés à l'annotation. Cette liste d'éléments, nommée *ListTGVObject*, contient des éléments de types différents composant le TGV. On peut distinguer les nœuds, les liens de nœuds, les contraintes, les liens de contraintes, les hyperliens (et leur héritage), les motifs d'arbre, le TGV et les annotations. Les annotations sont elles-mêmes considérées dans cet ensemble car elles permettent la composition des annotations (exemple : annotation d'algorithme annotée par une formule de coût).

Définition 4.8 : ListTGVObject

La *liste d'éléments* contient un ensemble d'éléments composant le TGV. Chacun de ces éléments est associé à l'annotation.

4.2.1.3 Annotation

Une annotation permet d'associer une liste d'élément à une information. Le type d'une annotation est défini par l'information associée. Ce type permet de grouper les annotations utilisées lors des vues 4.2.2.1.

Définition 4.9 : Annotation

Une *annotation* relie une *liste d'éléments* à une *information*. Le type de l'annotation est déterminée par l'*information* associée.

4.2.1.4 Ensemble d'annotations

L'ensemble d'annotations permet de définir un ensemble de bases d'annotation. Il contient toutes les annotations liées à un TGV (quelque soit le type d'information). Ainsi, l'optimiseur récupère les annotations dans cette liste pour les manipuler à l'aide de transformations. Il peut supprimer, modifier et ajouter une annotation de la liste.

Définition 4.10 : AnnotationSet

L'*ensemble d'annotations* est une liste de toutes les annotations d'un TGV.

4.2.2 Formalisation

Nous allons donc définir quatre Types Abstraits de Données correspondant aux définitions précédentes. Les annotations sont ajoutées à la définition des TGV ce qui permet d'enrichir le modèle pour intégrer toutes sortes d'information.

Information

L'information relative à l'annotation est donnée grâce à l'opération *getInformation* (définie par l'axiome *A1*). Cette information est définie à l'aide d'une chaîne de caractères. Toute extension de ce type abstrait pour créer une nouvelle annotation devra contenir l'information.

TA 36 Information I

Utilise : String

Opérations

O1	createInformation	String	→ I
O2	getInformation	I	→ String

Axiomes : $s \in String$;

A1 getInformation (createInformation (s)) = s

ListTGVObject

TA 37 ListTGVObject LT

Utilise : TGVElement E $\in$ {Node, NodeLink, Constraint, ConstraintLink, Hyperlink, TreePattern, TGV}, int

O1	createListTGVObject		→ LT
O2	addLT	LT × E	→ LT
O3	getLT	LT × int	→ E
O4	sizeLT	LT	→ int

Pré-conditions : $lt \in LT$; $i \in int$;

P1 define (getLT (lt, i)) $\Longleftrightarrow$ i >= 0 && i < sizeLT (lt)

Axiomes : $l \in LT$; $e \in E$;

A1	getLT (addLT (l, e), sizeLT (addLT (l, e)) - 1)	= e
A2	sizeLT (createListTGVObject)	= 0
A3	sizeLT (addLT (l, e))	= sizeLT (l) + 1

Le type abstrait 37 défini l'association d'une liste d'objet (*O1*), ainsi que toutes les opérations associées. L'opération *O2* permet d'ajouter un élément à la fin de la liste (*A1*) à laquelle s'associe l'opération de récupération d'élément *O3* dont l'indice détermine sa place dans la liste. Cet indice est borné entre 0 et la taille de la liste (*P1* et *P2*). La taille de la liste est définie dans l'opération *O4*, sa valeur est de zéro à la création (*A2*) et incrémentée de un à chaque ajout d'élément (*A3*). Ainsi, chaque élément ajouté par *addLT* peut être récupéré grâce à l'opération *getLT* grâce à son indice lors de son ajout.

Annotation

TA 38 Annotation A

Utilise : String, LT, I

Opérations

O1	createAnnotation	LT × I	→ A
O2	getAnnotation	A	→ I
O3	getListTGVObject	A	→ LT

Axiomes : $lt \in LT$; $a \in A$;

A1 getAnnotation (createAnnotation (lt, a)) = a

A2 getListTGVObject (createAnnotation (lt, a)) = lt

L'opération de création *O1* prend une liste d'élément et une annotation en para-

mètre. Les opérations *O2* et *O3* permettent de récupérer respectivement l'annotation et la liste d'élément associée. Celles-ci sont déterminées par leurs axiomes respectifs *A1* et *A2*.

AnnotationSet

TA 39 AnnotationSet AS

Utilise : A, int

O1	createAnnotationSet		→ AS
O2	addAS	AS × A	→ AS
O3	getAS	AS × int	→ A
O4	deleteAS	AS × int	→ AS
O5	sizeAS	AS	→ int

Pré-conditions : $l \in AS$; $i \in int$;

P1	define (getAS (l, i))	⟺ i >= 0 && i < sizeAS (AS)
P2	define (deleteAS (l, i))	⟺ i >= 0 && i < sizeAS (AS)

Axiomes : $l \in AS$; $a \in A$;

A1	getAS (addAS (l, a), sizeAS (addAS (l, a)) - 1)	= a
A2	deleteAS (addAS (l, a), sizeAS (addAS (l, a)) - 1)	= l
A3	sizeAS (createAnnotationSet)	= 0
A4	sizeAS (addAS (l, a))	= sizeAS (l) + 1
A5	sizeAS (deleteAS (l, i))	= sizeAS (l) - 1

La création d'un ensemble d'annotations est déterminée par l'opération *O1*. L'ajout d'une base d'annotation (*O1*) se fait à la fin de l'ensemble (*A1*). La récupération d'une base d'annotation est déterminée par l'opération *O3* dont l'indice détermine sa place, celui-ci doit être compris entre 0 et la taille de la liste. De même pour l'indice de suppression de base d'annotation (*O4*) défini par l'axiome *A2* redonnant l'état de la liste avant l'ajout d'une annotation. La taille de la liste est de zéro à la création (*A3*), incrémentée de un à chaque ajout (*A4*) et décrémentée de un à chaque suppression (*A5*).

4.2.2.1 Vues sur les annotations

Les annotations permettent d'associer une information à un ensemble d'éléments d'un TGV. Nous définissons une représentation de ces annotations pour nous permettre de visualiser celles-ci de manière intuitive. Pour cela, chaque annotation est associée à une couleur unique. L'annotation détermine donc la couleur associée des éléments annotés du TGV.

Pour représenter une annotation, nous associons la couleur à l'ensemble d'élément et la couleur est mis en légende du TGV. Comme il n'est pas possible d'associer plusieurs couleurs à un même élément, des *vues* sont définies pour chaque type d'annotation. Ces vues représentent un TGV avec un type d'annotation.

Dans le cas où plusieurs annotations sont nécessaires, il est possible d'utiliser une

notation à l'aide de chiffres (entre parenthèses) associé à l'élément considéré. Le chiffre est alors mis en légende avec l'annotation associée. Cette vue est très utile dans le cas où la couleur n'est pas possible (impression noir & blanc).

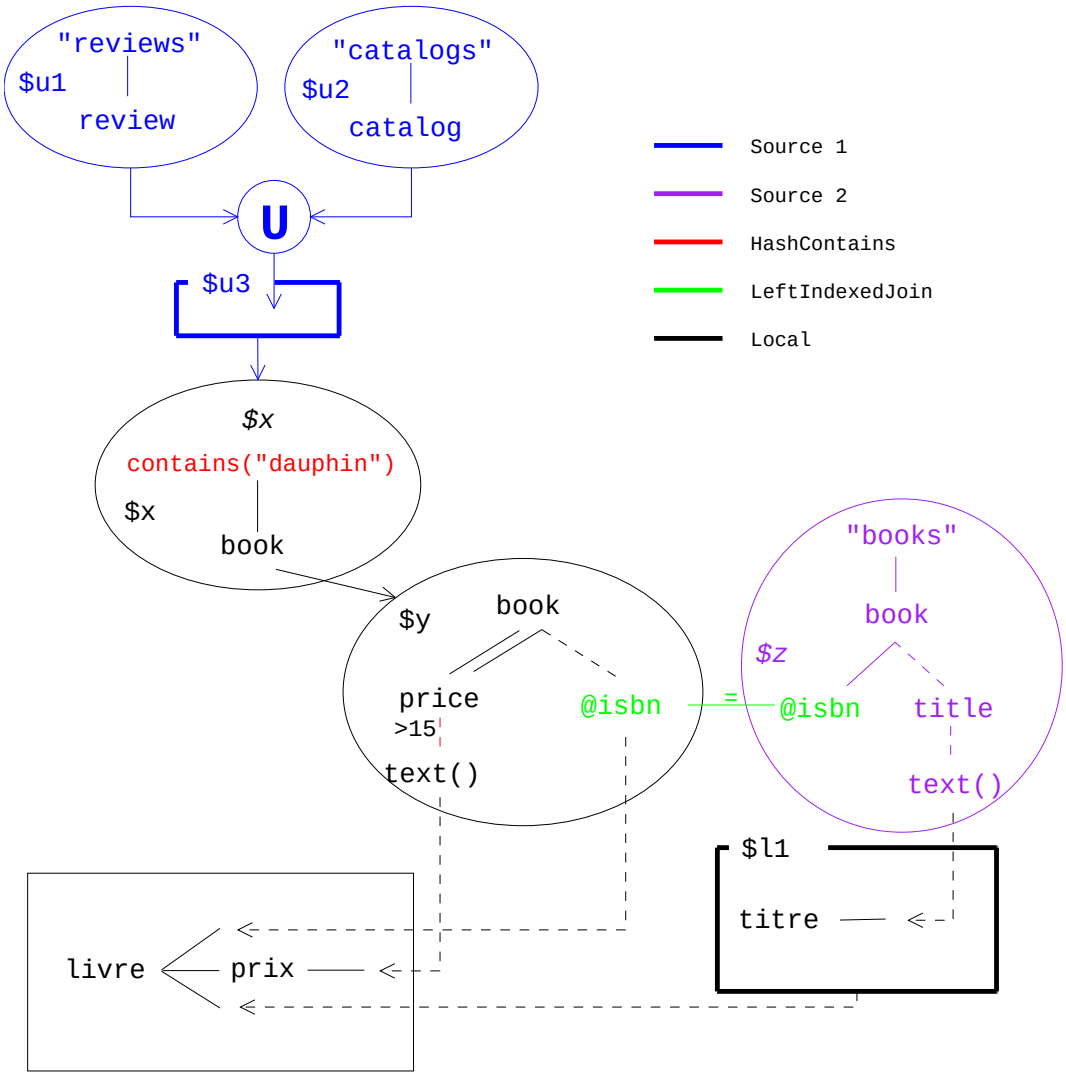


FIG. 4.2 – Exemple d'annotation de TGV

La figure 4.2 illustre plusieurs annotations physiques sur un TGV. Nous pouvons identifier cinq groupes de couleurs dans cet exemple. Comme ses annotations ne rentrent pas en concurrence, nous les avons associées dans une même vue.

Le premier ensemble en bleu représente la sélection des éléments correspondant à l'analyse de l'union de "*reviews*" et "*catalogs*" par la source 1, ainsi, l'évaluation de cette partie de tgv sera distante, et le résultat sera traité localement par les autres opérateurs.

Le second ensemble en violet détermine les résultats de la source 2 pour la collection

"*books*". Il est à noter que la couleur différente pour l'élément "*@isbn*" fait que la jointure ne sera pas traitée sur la source distante.

Le troisième ensemble en rouge "*contains ($x, "dauphins")*" défini l'opérateur *contains* comme une fonction de hachage sur l'élément courant. Ainsi, nous connaissons l'opérateur qui sera utilisé pour trouver le mot "*dauphin*" : *HashContains*.

Le quatrième ensemble en vert détermine l'algorithme de jointure entre les deux éléments "*@isbn*". Ainsi, l'élément de gauche sera indexé pour permettre de le retrouver plus efficacement lors de l'apparition d'une valeur identique à droite. L'opérateur *LeftIndexedJoin* permet ainsi de déterminer l'opérateur physique qui sera utilisé au sein de l'évaluateur.

Le cinquième ensemble en noir détermine l'ensemble des éléments qui seront traités localement sans informations particulières. Cet ensemble est donc composé d'opérateurs de l'algèbre simple et sera évalué localement.

Ainsi, nous avons représenté dans ce TGV annoté, un ensemble de localisation (Source 1 et Source 2) et des choix d'algorithme d'évaluation pour une fonction particulière (*contains*) et pour la jointure entre deux collections.

4.2.3 Conclusion

Les annotations nous permettent de définir, à un niveau de granularité quelconque, une annotation spécifique que nous pouvons définir aisément. Chaque information est annotée sur un ensemble d'éléments du TGV. Cette annotation est représentée par une couleur unique qui détermine, sur la représentation, l'ensemble des éléments associés. Il est alors aisé d'identifier les localisations des sources et les algorithmes choisis. Il est aussi possible d'ajouter des formules de coût.

Cette base d'annotation permet de manipuler les TGV a un niveau supérieur dans le cadre de l'optimisation. Un langage de définition de règles basé sur les TGV et les annotations transforme les TGV pour obtenir un plan optimal. Un *TGV logique* annoté est alors appelé un *TGV Physique*.

4.3 Règles de transformation

L'optimiseur doit pouvoir intégrer un ensemble de règles de transformations qui vont modifier le modèle TGV en fonction de sa structure correspondante à une requête *XQuery*. Ces règles s'appuient sur un ensemble de connaissances provenant de la logique des TGV, des sources dans le cas du contexte de médiation, des algorithmes d'opérateurs, etc. Cet ensemble permet de définir les règles de transformations. Ces règles peuvent préserver l'équivalence des résultats. Pour permettre à l'optimiseur

d'évoluer à chaque compilation, un langage de définition de règle est défini pour donner un *optimiseur extensible*.

4.3.1 Définition des transformations

Pour optimiser le modèle TGV, un ensemble de règles de transformation doit être défini dans l'optimiseur pour donner une représentation optimale d'une requête. Toutefois, l'ensemble des règles définies dans l'optimiseur peut varier au cours du temps et nécessiter des ajouts ou des suppressions, dépendant des besoins du contexte ou de l'administrateur du médiateur. Ainsi, un *optimiseur extensible* correspond à ces besoins, car en donnant un langage de définition de règles, l'optimiseur peut intégrer de nouvelles règles au système sans le modifier intégralement. Le langage de définition de règles permet ainsi d'intégrer de nouvelles transformations en utilisant simplement les définitions de notre modèle. Pour rester dans un modèle intuitif de représentation de requêtes à l'aide des TGV, un modèle de représentation de règles appelé *Rule Pattern* sera proposé pour permettre de visualiser la transformation créée.

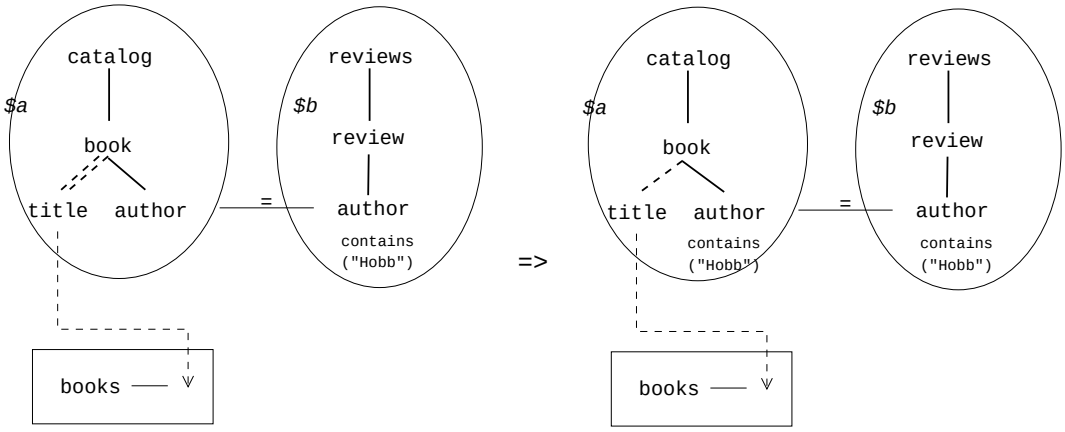


FIG. 4.3 – Transformation d'un TGV

L'exemple de la figure 4.3 représente sur le tgv de gauche une requête de jointure entre deux collections : «*catalog*» et «*review*». Une contrainte est liée à l'auteur de la revue qui doit contenir le mot «*Hobb*». Puis les deux noms d'auteur sont joins et enfin tous les titres contenus dans les livres sont retournés.

Le TGV de droite est une transformation équivalente du TGV de gauche après application de règles de transformation. Deux modifications sont notables, le lien de nœuds entre *book* et *title*, et la contrainte sur le nœud *author* de la collection *catalog*. La première transformation à modifier lien ancêtre/descendant en un lien parent/enfant, compte tenu des informations fournies par les sources, la balise *title* n'existe qu'à un seul endroit dans la collection *catalog*, et ainsi, le chemin a pu être complété. La seconde transformation correspond à une copie de contrainte à

travers l'équi-jointure, en effet, si l'auteur de la collection *review* doit contenir le mot «*Hobb*» et être égal à l'auteur de la collection *catalog*, alors, l'auteur de cette même collection doit contenir le mot «*Hobb*».

Ainsi, nous allons définir le langage de définition de règles de transformation pour nous permettre d'optimiser les TGV, et définir une représentation de ces règles de transformation.

4.3.1.1 Langage de règles

Un optimiseur est dit «*extensible*» si l'on peut l'enrichir à l'aide de définitions de nouvelles règles de transformation sans avoir à modifier le système. Pour cela, il nous faut définir un langage de définition de règles. Celui-ci doit pouvoir décrire la manipulation des TGV et l'utilisation des opérations de transformation.

Ainsi, le langage de définition de règles doit pouvoir modifier un TGV logique ou physique (avec annotations) pour déterminer la sélectivité de la transformation et la transformation à effectuer. Ainsi, le langage de définition de règles est décomposé en deux parties : la condition de la règle (ou *Rule Condition*) et le résultat de la règle (ou *Rule Conclusion*).

Puisque l'optimiseur travaille sur le modèle TGV, le langage de règle doit définir des opérations sur celui-ci. Les types abstraits de données (ADT) définissant les TGV permettent à la fois de définir les éléments à sélectionner, les conditions d'application de la règle, et les opérations de transformation. Il est donc naturel d'utiliser les types abstraits pour définir le langage de règles. De plus, l'expressivité du langage en sera d'autant plus grande que toute modification sur les TGV sera possible (définition précise de chaque élément). Alors, nous pouvons définir le langage de définition de règles par les trois clauses ci-dessous :

[Identifiant] :
FOR [Variables]
IF [Conditions]
THEN [Modifications]

Tab. 4.4 – Langage de définition de règles de transformation

Dans le tableau 4.4, nous pouvons voir la définition de notre langage en un identifiant de règle et trois clauses principales : *FOR, IF, THEN*. Chacune permet de définir une propriété de la règle :

- **FOR** : Variables d'entrée de la règle en fonction de son type définissant l'espace de recherche des éléments concernés. Ces types peuvent être : *Node, NodeLink, TreePattern, Hyperlink, Constraint, Annotation* ;

- **IF** (*Rule Condition*) : Condition d'application de la règle sur les éléments de la

clause *FOR*. Les opérations des types abstraits sont utilisées pour définir les conditions (grâce aux axiomes et aux pré-conditions), et des variables temporaires sont utilisées pour faciliter la manipulation, chacune d'elle est déclarée de la sorte : *$var := ...*;

– **THEN** (*Rule Conclusion*) : Transformations à appliquer dans le cas où les conditions de la clause *IF* sont vérifiées. Les variables temporaires peuvent être réutilisées pour modifier leurs valeurs associées. Les opérations de transformations sont celles définies dans les types abstraits (création, destruction, mise à jour, cf. Annexe D).

La règle de transformation *R1* du tableau 4.5 illustre le langage de définition de règles de transformation. Cette règle correspond à l'exemple donné dans la figure 4.3 dans laquelle la jointure a permis de copier la contrainte d'un nœud vers celui qui lui était lié par une équi-jointure.

	R1 :
1 :	**FOR** $\$H_1 \epsilon HJ, \$i \epsilon Integer$
2 :	**IF**
3 :	$\$n_1 := getLeftJH(\$H_1)$
4 :	$getConstraintName(getConstraint(\$H_1)) ==' ='$
5 :	$\$c := getConstraint(\$n_1, \$i)$
6 :	$\$n_2 := getRightJH(\$H_1)$
7 :	**THEN**
8 :	$createConstraint(\$n_2, \$c)$

Tab. 4.5 – Règle de transformation de distributivité de la jointure

La condition de la règle (ou *Rule Condition*) parcourt l'ensemble des hyperliens de jointure auquel est associé un entier (ligne 1). Pour cet ensemble d'hyperliens, chacun d'eux est associé à une variable $\$H_1$, et les entiers à la variable $\$i$. Les lignes 2 à 6 définissent la sélectivité de la règle à l'aide des opérations des types abstraits des TGV. A la ligne 3, la variable $\$n_1$ définit le nœud gauche de l'hyperlien de jointure, alors que le nœud de droite est associé à la variable $\$n_2$à la ligne 6. Ensuite, la contrainte de jointure (*getConstraint*) doit être un opérateur d'égalité (ligne 4). Enfin, le nœud $\$n_1$ (celui de gauche) doit être associé à une contrainte pour l'entier $\$i$ (ligne 5) et associée à la variable $\$c$. Si ces conditions sont vérifiées, c'est à dire si la jointure contient l'opérateur '=' et s'il existe une contrainte $\$c$ associée à $\$n_1$ à la position $\$i$, alors, la conclusion de la règle peut être appliquée. Celle-ci doit pouvoir copier la contrainte du nœud de gauche au nœud de droite. Ainsi, à la ligne 8, l'opération *createConstraint* associe la variable $\$c$ définie à la ligne 5 au nœud $\$n_2$ défini à la ligne 6.

Ainsi, à l'aide de l'expressivité des types abstraits de données définis pour les TGV et de leurs annotations, le langage de règle est capable d'exprimer un large ensemble de règles de transformation sur les TGV et ainsi de les optimiser. Par la suite, nous allons étudier une représentation de ces règles en utilisant celle définie pour les TGV pour représenter un motif à appliquer sur ceux-ci.

4.3.1.2 Motifs de Règles

Maintenant que nous avons un langage de règle permettant de définir des transformations sur les TGV, nous proposons un modèle de représentation de règles de transformation : *les motifs de règles* (ou *Rule Patterns*). Comme le modèle TGV représente un mapping entre des documents XML et un ensemble de motifs d'arbre, les motifs de règles seront représentés comme un motif applicable sur un TGV. Ainsi, nous avons une représentation intuitive des règles de transformation.

Comme la représentation des *TGV* provient de sa formalisation à base de types abstraits de données, nous pouvons utiliser naturellement cette représentation pour les motifs de règles, mais spécialisée à certaines parties du TGV. Les différences entre le modèle *TGV* et le modèle *Rule Patterns* viennent des variables définies dans la condition de la règle, et les éléments sélectionnés par la clause *FOR*. Pour représenter ces motifs de règles, chaque élément déclaré dans les clauses *FOR* et *IF* sera représenté. Nous leur associerons les variables déclarées dans la clause *IF*. Les opérations de comparaisons de type '==' seront représentés par leurs valeurs de comparaison pour montrer ce que le motif requièrent pour être sélectionné.

R : **Motif de règle condition** => **Motif de règle conclusion**

Tab. 4.6 – Représentation d'une règle de transformation

Deux motifs sont nécessaires pour représenter une règle de transformation. En effet, le premier motif appelé «*motif de règle condition*» (ou *Condition Rule Pattern*) doit représenter les conditions d'application de la règle des clauses *FOR* et *IF*. Le second motif «*motif de règle conclusion*» (ou *Conclusion Rule Pattern*) représente l'état final après application de la règle en utilisant les variables définies dans le motif précédent. Les différences entre les deux motifs de règles correspondent aux modifications appliquées dans la clause *THEN*. Le tableau 4.6 donne la représentation d'une règle de transformation avec le nom de la règle tout à gauche, le motif de règle condition à gauche et le motif de règle conclusion à droite.

Fig. 4.4 – Motifs de Règles de distributivité de la jointure

Afin d'illustrer les motifs de règles, la règle définie dans le tableau 4.5 est représentée dans la figure 4.4. Le motif de gauche représente le *motif de règle condition* ou *Condition Rule Pattern*, avec un hyperlien de jointure annoté par la variable $\$H_1$, la contrainte de jointure est associée à la jointure par le symbole '=' puisqu'il correspond à la valeur qu'il doit avoir dans le motif de règle. La variable $\$n_1$ correspond au noeud gauche de l'hyperlien et remplace le nom du nœud qui n'est pas spécifié. La

contrainte $\$c$ est associée à la variable $\$n_1$ sous celle-ci, comme pour une contrainte liée à un nœud. Enfin, la variable $\$n_2$ correspond au noeud droit de l'hyperlien.

Le motif de règle de droite représente le *motif de règle conclusion* ou *Conclusion Rule Pattern*. Il est obtenu après transformations sur le motif de règle condition. Ici, la contrainte $\$c$ reliée au nœud $\$n_1$ est dupliquée (même nom de variable) et associée au nœud $\$n_2$ de l'autre côté de l'hyperlien. Cette règle de transformation permet de réduire le nombre de résultats produits dans l'ensemble de droite grâce à la nouvelle contrainte associée (de plus amples détails sont donnés dans la section suivante). Ainsi, nous obtenons grâce aux deux motifs de règles un motif global permettant de définir visuellement une règle de transformation que nous pouvons appliquer sur un TGV.

4.3.1.3 Conclusion

Nous avons défini un langage de définition de règles de transformation applicable sur les TGVs, auquel nous avons associé de la même manière que les TGV un modèle de représentation : *Rule Patterns*. Ces représentations sont des motifs que nous pouvons appliquer sur un TGV et ainsi nous permettre de les optimiser.

Le langage de règles de transformations est basé sur les types abstraits de données. Ce choix nous permet de définir des règles de transformation très précises sur les modifications effectuées sur les TGV. De plus, la méthode de représentation des TGV à partir des types abstraits facilite la représentation des motifs de règles.

Par la suite, nous avons besoin d'orienter la stratégie de recherche définie dans l'optimiseur, à cette fin, une classification des règles de transformation est nécessaire pour définir différents ensembles de transformations.

4.3.2 Classification des transformations

Les règles de transformation définies grâce au langage de règles permettent à l'optimiseur extensible d'améliorer sa base de connaissance sur les TGV. Ainsi, de nouvelles modifications des TGV sont intégrées et permettent d'atteindre la représentation optimale.

Ces transformations nécessitent une classification pour nous permettre d'orienter la stratégie de recherche. Dans l'ensemble des transformations possibles sur les TGV, trois classes caractéristiques peuvent être définies. Les transformations logiques équivalentes, les transformations physiques équivalentes et les transformations utilisateurs.

Les transformations logiques équivalentes sont des transformations qui reposent uniquement sur la connaissance du modèle. Elles modifient celui-ci pour donner la

représentation la plus optimale possible sans utiliser les connaissances externes au modèle (informations physiques). Elles sont dites équivalentes car elles préservent le résultat de l'évaluation quelque soit la transformation.

Les transformations physiques équivalentes définissent des transformations qui utilisent les connaissances externes au modèle, comme les capacités des sources, les métadonnées, les choix d'algorithme d'opérateurs... Ces transformations sont aussi équivalentes car le résultat reste identique.

Les transformations utilisateurs sont des transformations externes définies par l'administrateur. Elles correspondent à une connaissance propre au contexte dans lequel s'intègre l'optimiseur. Ainsi, ces transformations ne peuvent être appelées équivalentes car rien ne garantit l'équivalence des résultats de l'évaluation. Toutefois, ces transformations permettent de répondre à un contexte particulier demandant des modifications significatives des requêtes.

Comme nous pouvons le voir, certaines transformations sont dites équivalentes, il est donc nécessaire de définir l'équivalence entre deux représentations, et l'équivalence d'une règle de transformation.

4.3.2.1 Équivalence

Un optimiseur de requêtes utilise des règles de transformations pour modifier celle-ci pour trouver celle dont l'évaluation sera la plus optimale. Toutefois, cet optimiseur doit pouvoir garantir un résultat identique après réécriture de la requête lors de l'application de la règle de transformation. Dans ce cas, les deux requêtes sont dites équivalentes.

Définition 4.11 : Équivalence

Deux TGV tgv_1 et tgv_2 sont dits équivalents s'ils ont le même résultat lors de l'évaluation, quelque soit l'ensemble d'arbres τ :

$$\forall\ \tau, eval(tgv_1, \tau) = eval(tgv_2, \tau)$$

Une règle de transformation modifie un TGV pour en obtenir un nouveau. Grâce à la définition de l'équivalence, nous pouvons alors définir qu'une règle de transformation est dite correcte si elle préserve l'équivalence des résultats après application de la transformation.

Définition 4.12 : Règle correcte

Une règle de transformation φ est dite correcte si elle préserve l'équivalence du TGV transformé :

$$eval(\varphi(tgv), \tau) = eval(tgv, \tau)$$

Ainsi, les règles de transformations sont dites correctes si elle préserve l'équivalence entre deux TGV. Alors, leurs évaluations doivent être identiques après modification. Ce qui veut dire que le résultat de la fonction *eval* doit être préservé par la règle de transformation, et doit donc contenir les mêmes ensembles d'arbre. Chaque règle ajoutée à l'optimiseur devra être prouvée pour qu'elle soit classée parmi les règles de transformations correctes.

Cette thèse ne prouve pas si une règle est correcte ou non. Un ensemble de règles correctes ont déjà été prouvées dans la littérature [Cherniack et Zdonik 1998] [Ali et Moerkotte 2004]. Il faut intégrer les règles existantes dans l'optimiseur.

Nous allons donc étudier les différentes catégories de règles de transformation : équivalentes logiques et physiques, et utilisateurs. Nous illustrons chacune d'elle par un exemple et un motif de règles.

4.3.2.2 Transformations logiques équivalentes

Les transformations logiques équivalentes sont des règles de transformation qui préservent l'équivalence et qui utilisent la connaissance sur les TGV pour le modifier. Celles-ci ne concernent donc que les améliorations topologiques sur les TGV, aucune connaissance extérieure n'est nécessaire pour ces transformations. Les transformations logiques ne manipulent pas les annotations, car celles-ci donnent un TGV Physique.

L'exemple de la règle *R1* définie dans le tableau 4.5 (section 4.3.1.1) associée à la figure 4.4 est une transformation logique équivalente. En effet, transférer une contrainte à travers une équi-jointure préserve l'équivalence car l'opérateur '=' est distributif. En effet, la sélectivité de l'ensemble d'arbre reste identique après évaluation (la jointure supprime ceux dont la valeur ne respecte pas la contrainte de $\$n_1$). De plus, cette transformation est de type logique car elle ne tient compte que des éléments présents dans les TGV logiques, aucune annotation extérieure n'est utilisée.

4.3.2.3 Transformations physiques équivalentes

Les transformations physiques équivalentes sont des transformations qui préservent l'équivalence tout en utilisant comme base de connaissance celles des TGV et celles des annotations extérieures (section 4.2). Ainsi, les informations physiques prove-

nant des sources comme les méta-données, les informations de coût, les capacités fonctionnelles, les choix d'algorithme, le contexte de médiation, etc... peuvent être utilisés par les transformations physiques équivalentes pour améliorer les TGV Physiques. Afin d'illustrer ce concept, nous proposons deux exemples de transformations physiques équivalentes : la permutation de la jointure et la modification d'algorithme de jointure.

Permutation de la jointure Lorsqu'une requête contient une jointure entre deux ensembles d'arbres, il peut être intéressant de définir un ordre pour effectuer la jointure. En effet, l'ordre d'exécution de la jointure influe sur les performances d'évaluation dépendant de la cardinalité de chacun des ensembles liés par la jointure. Dans le cas où les sources fournissent des informations de cardinalité des éléments, l'optimiseur doit être capable de connaître la meilleure manière d'évaluer la jointure.

La règle de transformation *R2* du tableau 4.7 illustre la permutation de la jointure. La clause FOR prend l'ensemble des hyperliens de jointure du TGV ($\$JH$). La clause IF vérifie tout d'abord si l'annotation d'algorithme de la jointure est une boucle imbriquée (BI). L'annotation de jointure est récupérée (ligne 3, 4, 5 et 6), puis la vérification est effectuée (ligne 7). Puis, la règle identifie les nœuds de gauche et de droite de la jointure (ligne 8 et 9). Pour ceux-ci, leurs statuts doit être obligatoire (ligne 10 et 11). De plus, la jointure doit être une équi-jointure pour que celle-ci ne modifie pas le résultat (ligne 12). Des lignes 13 à 19 (et de 20 à 26) sont vérifiés l'ensemble des annotations associées aux nœuds (ligne 13 et 20), puis pour chacune d'elles, cette annotation doit être de type cardinalité (ligne 17 et 24). Dans ce cas, la cardinalité est assignée à la variable $\$card_x$ (ligne 18 et 25). Enfin, la règle doit vérifier si la cardinalité a été assignée dans les deux cas (ligne 19 et 26), dans le cas contraire, la règle n'est pas appliquée. La ligne 27 donne la dernière condition de permutation de la jointure, si la cardinalité de $\$n_2$ est inférieure à celle de $\$n_1$, alors la règle est enfin appliquée. La clause $THEN$ fait alors permuter les deux éléments de la jointure (ligne 29 et 30).

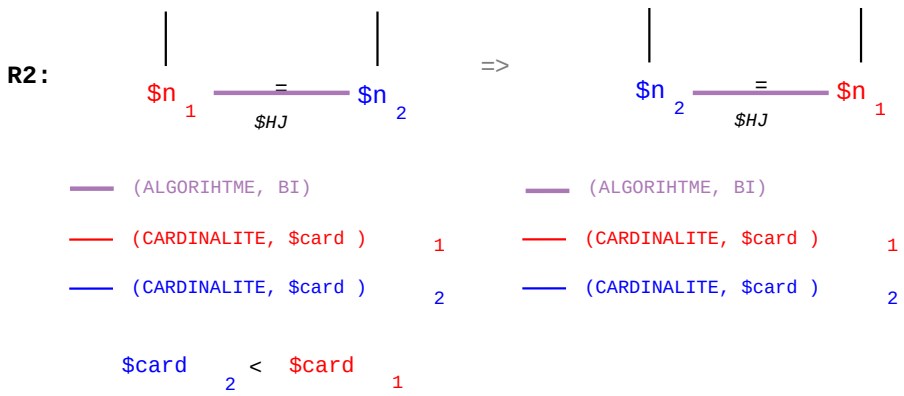


FIG. 4.5 – Motif de règle physique de permutation de jointure

```
      R2 :
1 :   FOR $JH ϵ JoinHyperlink, $AS ϵ AnnotationSet
2 :   IF
3 :      $as1 := getListAnnotation($AS, $n1)
4 :      FOR $i in 0 .. sizeAS($as1)
5 :         $a := getAS($as1, $i)
6 :         getAnnotation($a) instanceof "ALGORITHME"
7 :         (getInformation(getAnnotation($a)) == "BI")

8 :      $n1 := leftJH($JH)
9 :      $n2 := rigthJH($JH)
10 :     isMandatory(getNodeLink($n1)) == true
11 :     isMandatory(getNodeLink($n2)) == true
12 :     getConstraintName(getConstraint($JH)) ==''=''

13 :     $as1 := getListAnnotation($AS, $n1)
14 :     $card1 := −1
15 :     FOR $i in 0 .. sizeAS($as1)
16 :        $a := getAS($as1, $i)
17 :        getAnnotation($a) instanceof "CARDINALITE"
18 :        $card1 := getInformation(getAnnotation($a))
19 :     $card1! = −1

20 :     $as2 := getListAnnotation($AS, $n2)
21 :     $card2 := −1
22 :     FOR $i in 0 .. sizeAS($as2)
23 :        $a := getAS($as2, $i)
24 :        getAnnotation($a)instanceof"CARDINALITE"
25 :        $card2 := getInformation(getAnnotation($a))
26 :     $card2! = −1

27 :     $card2 < $card1
28 :  THEN
29 :     $setLeftJH($JH, $n2)
30 :     $setRightJH($JH, $n1)
```

TAB. 4.7 – Règle de transformation physique de permutation de jointure

La figure 4.5 représente le motif de règle correspondant à la règle de transformation du tableau 4.7. Nous observons, dans le motif de règle condition, les variables déclarées dans les différentes clauses définissant la jointure, les noeuds, leur ascendance obligatoire et les annotations de cardinalité (couleurs rouge et bleu) et d'algorithme de boucle imbriqué (violet). Nous pouvons voir que dans le cas où la cardinalité de $\$n_2$ est inférieure à $\$n_1$, alors la jointure est permutée. Le motif de règle conclusion montre la permutation des deux nœuds de l'hyperlien de jointure.

Algorithme de la jointure Pour évaluer un TGV, il faut pouvoir définir à l'aide de l'algèbre des algorithmes. L'exemple que nous présentons ici permet de modifier un algorithme de jointure. Il transforme une jointure par boucle imbriqué (*BI*) en algorithme de jointure par hachage (*HJ*). Ainsi, la règle de transformation manipule des annotations de type *Algorithme* et des annotations de type *Cardinalité* pour vérifier dans quel cas de jointure nous aurons à modifier l'algorithme.

La règle de transformation *R3* du tableau 4.8 illustre la modification de l'alogirhtme de jointure. Le principe est identique à la permuation à ceci prêt que l'on garde l'annotation de jointure dans la variable $\$aj$ (ligne 10). Les lignes 7 à 10 définisse une condition d'entrée dans la règle. Si la condition de la ligne 8 est vérifiée, alors on applique la ligne 10, sinon on passe à l'occurrence suivante. Pour la suite, la condition de la ligne 30 change par rapport à la permutation. En effet, une jointure par hachage est plus intéressante qu'une boucle imbriquée si la cardinalité du résultat de la jointure R1 x R2 est plus faible que la vérification directe R1 * R2. La cardinalité du résultat de la jointure est calculée grâce le modèle de coût à l'aide de l'opération $card()$. Ainsi, la formule «$\$card(card_1 \times \$card_2) < (card_1 * card_2)$» vérifie si la modification de l'algorithme est intéressante. Dans ce cas, l'algorithme est modifié à la ligne 32.

```
     R3 :
1 :  FOR $JH ϵ JoinHyperlink, $AS ϵ AnnotationSet
2 :  IF
3 :     $as1 := getListAnnotation($AS, $n1)
4 :     FOR $i in 0 .. sizeAS($as1)
5 :        $a := getAS($as1, $i)
6 :        getAnnotation($aj) instanceof "ALGORITHME"
7 :        IF
8 :           getInformation(getAnnotation($a) == "BI")
9 :        THEN
10 :          $aj := $a

11 :    $n1 := leftJH($JH)
12 :    $n2 := rigthJH($JH)
13 :    isMandatory(getNodeLink($n1)) == true
14 :    isMandatory(getNodeLink($n2)) == true
15 :    getConstraintName(getConstraint($JH)) ==′=′

16 :    $as1 := getListAnnotation($AS, $n1)
17 :    $card1 := −1
18 :    FOR $i in 0 .. sizeAS($as1)
19 :       $a := getAS($as1, $i)
20 :       getAnnotation($a) instanceof "CARDINALITE"
21 :       $card1 := getInformation(getAnnotation($a))
22 :    $card1! = −1

23 :    $as2 := getListAnnotation($AS, $n2)
24 :    $card2 := −1
25 :    FOR $i in 0 .. sizeAS($as2)
26 :       $a := getAS($as2, $i)
27 :       getAnnotation($a)instanceof"CARDINALITE"
28 :       $card2 := getInformation(getAnnotation($a))
29 :    $card2! = −1

30 :    $card(card1 × $card2) < (card1 ∗ card2)
31 :  THEN
32 :     $modifyAnnotationInformation($aj, "HJ")
```

TAB. 4.8 – Règle de transformation physique pour l'algorithme de jointure

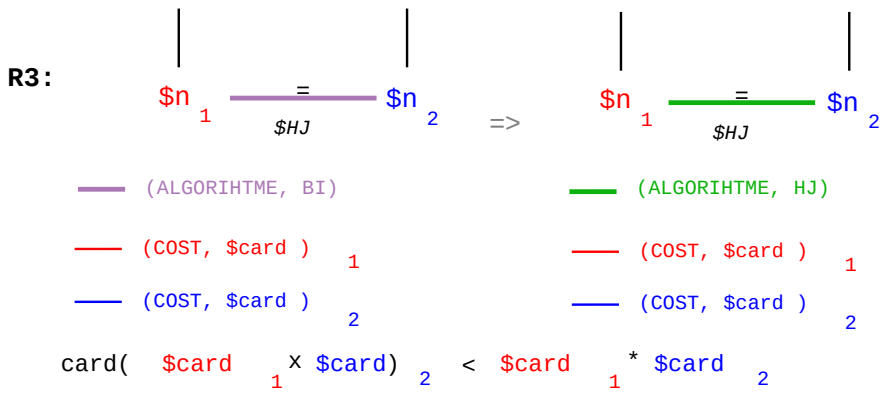


FIG. 4.6 – Motif de règle physique de modification d'algorithme de jointure

La figure 4.6 représente le motif de règle correspondant à la règle de transformation du tableau 4.8. Nous observons, dans le motif de règle condition, les variables déclarées dans les différentes clauses définissant la jointure, les noeuds, leur ascendance obligatoire et les annotations de cardinalité et d'algorihtme de jointure. La vérification des annotations est représentée en dessous à l'aide de la formule correspondante. Le motif de règle conclusion montre la modification de l'algorithme de jointure (couleur différente et information différente).

4.3.2.4 Transformations Utilisateurs

Les règles de transformation utilisateurs sont des règles basées sur une connaissance du système et des sources que seul l'administrateur peut connaître. Ainsi, ces informations externes aux TGV et annotations ne peuvent préserver l'équivalence du résultat. En effet, celles-ci sont très spécifiques et ne peuvent prouver si une règle est correcte dans un contexte global. Ainsi, nous classons ces règles arbitrairement définies par l'administrateur comme des règles utilisateurs. Ces transformations peuvent aussi bien être logiques (sans annotations) que physiques (avec annotations).

Il est possible d'intégrer des transformations utilisateurs dans l'optimiseur, malgré le fait que cela puisse introduire des différences dans les résultats obtenus. Par contre, cette possibilité est très intéressante pour répondre à l'adaptabilité de l'optimiseur en fonction du contexte dans lequel il est intégré. Prenons l'exemple de règles sémantiques. Celles-ci peuvent modifier le schéma un motif d'arbre. Chaque nom de nœud peut être modifié, voir de nouveaux liens ajoutés. Dans ce cas, le résultat n'est pas la même puisque la requête n'interroge plus les mêmes données. Par contre, la réponse peut correspondre à la requête de l'utilisateur de manière sémantique. Tout dépend du contexte défini pour l'intégration de l'optimiseur. Il est toutefois conseillé de bien vérifier les transformations intégrées à l'optimiseur.

```
     R4 :
1 :  FOR $N ϵ Node, ß, $j ϵ int
2 :  IF
3 :     label($N) == "book"
4 :     $n := getNextNode(getNextNodeLink($N, $i))
5 :     label($n) == "author"
6 :     $c := getConstraint(getConstraintLink($n, $j))
7 :     $c instanceof "Function"
8 :     getFunction($c) == "contains"
9 :     getParam(getParameters($c), 0) == "RobinHobb")
10 : THEN
11 :    setLabel($N, "RobinHobb")
12 :    removeConstraint($n, $c)
13 :    removeNode($N, $n)
```

TAB. 4.9 – Règle des transformation utilisateur changeant le nom d'un label

La règle de transformation *R4* définie dans le tableau 4.9 illustre une règle de transformation utilisateur. En effet, la règle vérifie si le TreePattern cherche l'auteur «Robin Hobb», dans ce cas, le nœud d'origine est modifié, tandis que le nœud et la contrainte sont supprimés. La clause FOR prend l'ensemble des nœuds du TGV ainsi que de nombres quelconque $\$i$ et $\$j$. Dans la clause IF, si le nœud choisi n'a pas pour nom de label «book» la règle n'est pas vérifiée (ligne 3). Puis, le $\$i^{\circ}$ nœud fils doit avoir un nom de label «author» (ligne 4 et 5). Ensuite, la $\$j^{\circ}$ contrainte doit être une fonction «contains» (ligne 6, 7 et 8) et doit rechercher le mot «Robin Hobb» (ligne 9). Dans ce cas, la règle doit être appliquée. La clause $THEN$ applique la règle en modifiant le nom de label de $\$N$ (ligne 11), supprime la contrainte (ligne 12) et le nœud «author» (ligne 13).

Cette règle de transformation utilisateur exprime une transformation non équivalente mais pouvant exprimer une transformation sémantique correspondant aux données réellement existantes sur les sources de données. Avant la transformation, la requête ne pouvait peut-être pas retourner de réponse, tandis que pour un motif d'arbre différent (après transformation) la requête peut alors retourner les résultats correspondants à cette nouvelle requête. Tout dépend du contexte dans lequel s'intègre l'optimiseur et de la définition que l'on souhaite donner aux règles de transformation.

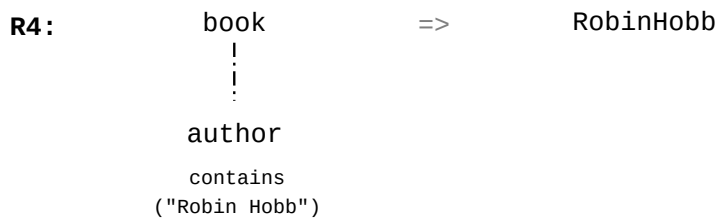


FIG. 4.7 – Motif de règle utilisateur de changement de nom de label

La figure 4.7 illustre la règle du tableau 4.9. Le motif de règle condition requière les noms de nœuds et la fonction «contains» associée. Le motif de règle conclusion donne alors le nouveau motif d'arbre, avec le nom de label modifié et la suppression de la fonction et du nœud «author».

4.3.3 Conclusion

Les règles de transformations peuvent être classées dans trois groupes de transformations : équivalentes logiques, équivalentes physiques et utilisateurs. Les transformations équivalentes sont des transformations qui modifient un TGV sans modifier le résultat de son évaluation, une transformation logique est une transformation qui n'utilise que les connaissances internes aux TGV, alors qu'une transformation physique utilise les connaissances des TGV et de leurs annotations. Enfin, les transformations utilisateurs sont des transformations dont la base de connaissances ne peut être annotée, de plus, les résultats de ces transformations ne sont pas forcément équivalents mais répondent aux pré-requis du contexte définis lors de la conception de l'optimiseur.

Maintenant que nous avons défini des catégories de classification, nous allons pouvoir définir une stratégie de recherche pour la génération de TGV. Cette stratégie utilise les transformations pour atteindre le TGV optimal, les classes de transformations pourront être utilisées pour orienter cette génération de TGV.

4.4 Stratégie de recherche

La stratégie de recherche permet d'atteindre l'évaluation d'une requête optimale. Ainsi, grâce à cette stratégie, le choix des règles de transformation est orienté vers la génération d'une requête la plus optimale possible. Ce qui revient à dire que nous cherchons à obtenir un TGV annoté optimal, puisque celui-ci correspond à l'évaluation de la requête définie par l'algèbre abstraite (section 4.1).

Cette stratégie de recherche doit pouvoir quantifier chaque TGV pour lui permettre de connaître, pour chaque représentation, le coût de son évaluation. Cette quantification s'appuie sur un modèle de coût (section 4.4.1) utilisant les annotations définies précédemment (section 4.2). Le choix de la règle de transformation s'appuie sur la génération d'un espace de recherche (section 4.4.2), permettant de choisir le plus rapidement possible les transformations les plus efficaces.

Le choix de la meilleure stratégie d'optimisation n'est pas le sujet de cette thèse. En effet, nous proposons un cadre d'optimisation permettant d'optimiser les TGV. Une simple stratégie de recherche incrémentale est utilisée par la suite. Cette stratégie de recherche sera ultérieurement améliorée dans la thèse de [Liu to appear], dans lequel un modèle de coût pour les TGV est défini auquel sera adapté une nouvelle stratégie de génération de TGVs. La stratégie de recherche utilisée nous permet d'effectuer des tests de performances validant le fonctionnement de l'optimiseur extensible (section 5.5).

4.4.1 Modèle de coût

Le modèle de coût permet de donner un coût d'évaluation théorique pour une requête. Il permet de donner une différence entre l'évaluation directe et l'évaluation finale par l'intermédiaire de règles de transformations, ces règles font varier le coût des TGV et orientent l'optimiseur. Ainsi, le modèle de coût est associé aux TGV pour donner un coût à chaque TGV permettant d'atteindre le TGV optimal, celui dont le coût sera le moindre parmi tous les TGV pouvant être généré par les règles de transformations.

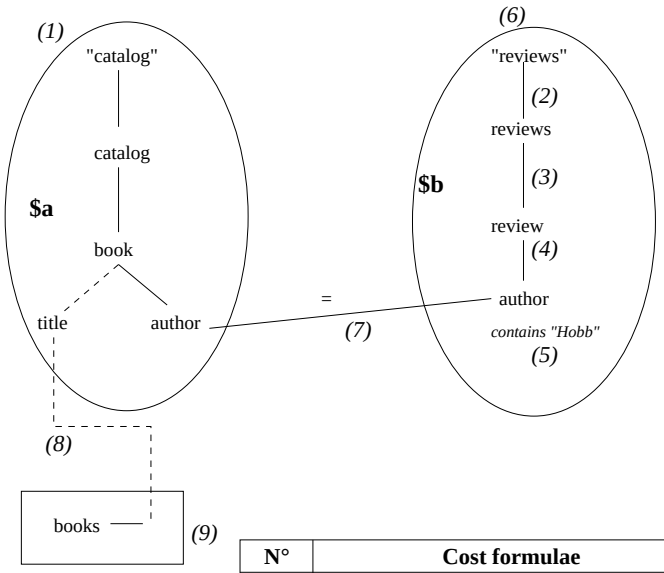
Dans le cadre de la médiation, un modèle de coût doit pouvoir intégrer les problèmes d'hétérogénéité des sources et de l'évaluation distantes. En effet, chaque source apportant une évaluation très hétérogène, il est difficile de se reposer sur un simple historique. De plus, un historique pourrait être trop variable entre deux évaluations obtenues par règle de transformation. Ainsi, notre modèle de coût repose sur un modèle de coût de médiation qui est capable d'approximer de façon plus efficace le coût de l'évaluation d'une requête distante et ainsi affecter le choix d'une règle.

Le modèle de coût proposé par Tianxao Liu [Liu to appear], dans le cadre de sa thèse de recherche autour des TGV, intègre les problèmes de médiation, les opérations de l'algèbre abstraite définies pour les TGV et les opérateurs définies dans le médiateur pour permettre de donner une valeur à chaque TGV.

Le modèle de coût est représenté par un système d'équations associé à un ensemble d'éléments du modèle. Chaque équation correspond à la fonction de coût qui peut être définie par les sources ou le médiateur, elle peut contenir des informations statistiques sur les sources, des variables de coût ou des formules de coût. Puisque les annotations des TGV permettent d'intégrer toutes sortes d'informations, chaque équation pourra être associée à un ensemble d'éléments via l'annotation de TGV. Un nouveau type d'annotation (par héritage) sera alors créé : *CostAnnotation* (ou annotation de coût), permettant de contenir une équation du modèle coût. Chaque annotation de coût sera associée à un ensemble d'éléments représentant l'équation donnant le coût de l'évaluation de cet ensemble.

Le modèle de coût utilisé pour le modèle TGV est basé sur le calibrage des sources raffiné par un historique d'informations. Pour obtenir des modèles de coût pour notre modèle, une procédure de calibrage [Du *et al.* 1992][Gardarin *et al.* 1996] par exécution de quelques requêtes de calibrage à l'initialisation du système. Des informations de coût sont obtenues (par exemple des statistiques sur les sources) par ce calibrage, mais celles-ci doivent pouvoir être maintenues à jour au cours du temps. Ainsi, durant l'exécution de requêtes par le médiateur sur les sources, le coût de l'évaluation de chaque requête distante est stockée dans une relation entre le type de la requête et son coût d'exécution [Adali *et al.* 1996]. Cet historique permettra alors de raffiner le modèle de coût initial et de le remettre à jour.

Pour connaître le coût d'évaluation d'un TGV, l'ensemble des informations de coût annotées sur ce TGV est regroupé dans un système d'équation qui pourra être traité



card: cardinality

sel: selectivity

restr: restriction

proj: projection

N°	Cost formulae
(1)	Cost_s1=2*cost_unitary_s1
(2)	Cost_link_reviews = Cost_IO * card (reviews)
(3)	Cost_link_review = Cost_IO*card(catalogs)
(4)	Cost_link_author = 2*card (catalogs)*sel
(5)	Cost_restr_author = Cost_IO*sel
(6)	Cost_s2=(1)+(2)+(3)
(7)	Cost_join=Cost_CPU*(card_s1*card_s2)
(8)	Cost_proj=result_card
(9)	Cost_global=Max((1), (5))+(7)+2*(6)

FIG. 4.8 – TGV annoté par des formules de coût

globalement pour donner le coût d'évaluation de cette requête.

La figure 4.8 illustre l'annotation d'un TGV à l'aide des informations de coût définies, en *MathML*, dans le médiateur pour chaque ensemble d'éléments. Nous pouvons voir qu'une annotation de coût (1) est définie pour le motif d'arbre $\$a$, la formule correspondante est donnée dans le tableau "*Cost formulae*". L'ensemble des annotations de coûts (2, 3, 4, 5) du motif d'arbre $\$b$ donne un coût global (6). L'hyperlien de jointure est annoté par le coût (7) avec la formule correspondante à la jointure interne au médiateur. Puis la valeur de projection du titre est donnée par la formule (8). Enfin, la construction du résultat est annotée avec la formule (9) ayant pour coût global le maximum du coût d'évaluation de (1) et (5) auquel est ajouté le double du coût d'évaluation de (6). Ainsi, grâce à l'ensemble des formules de coût annotées sur le TGV, nous pouvons connaître son coût d'évaluation théorique.

Le modèle de coût défini pour les TGV permet donc de définir le coût d'évaluation de chaque TGV par le médiateur. L'optimiseur pourra alors connaître le coût de chaque TGV généré à l'aide des règles de transformation et ainsi choisir le TGV le plus

optimal parmi ceux proposés par l'optimiseur. Il est donc nécessaire de définir une stratégie de génération de TGV pour orienter le choix des règles de transformation et réduire l'espace de recherche de TGV.

4.4.2 Espace de Recherche

L'espace de recherche est un ensemble de TGV pouvant être généré à l'aide des règles de transformations définies dans l'optimiseur. Si aucune stratégie de recherche n'est définie dans l'optimiseur, alors, l'espace de recherche génère l'ensemble de tous les TGV possible, le TGV optimal est alors celui dont le coût d'évaluation, défini par le modèle de coût, est le moins cher. Toutefois, la génération de tous les TGV est souvent plus longue que l'évaluation du TGV initial, il n'est donc pas utile de générer tout l'espace de recherche. C'est pour cela qu'une stratégie de recherche doit être définie pour réduire l'espace de recherche et ainsi obtenir un TGV optimal dit "local" à l'espace généré. Ainsi, la stratégie de recherche oriente le choix d'utilisation des règles de transformation pour réduire l'espace de recherche pour obtenir le TGV optimal local.

L'optimiseur *EXODUS* [Carey *et al.* 1990] propose une stratégie de recherche basée sur la valuation des règles de transformations avec un coefficient d'amélioration. Ce coefficient est défini entre 0 et 1 par un calibrage des règles à l'initialisation du système, plus le coefficient est proche de zéro, plus le gain de performance est grand, plus il est proche de 1, plus le gain est petit (coût identique à l'initial). Ainsi, l'optimiseur choisit, pour une requête donnée, la règle de transformation dont le coefficient d'amélioration est le plus proche de zéro, dans l'ensemble des règles applicables sur cette requête. La stratégie de recherche incrémentale n'est certes pas la plus efficace puisqu'elle ne génère pas forcément le meilleure plan. Toutefois, nous avons choisi cette solution en premier lieu pour établir un cadre d'optimisation nous permettant de tester les performances de l'optimiseur extensible. La stratégie de recherche peut être modifiée pour améliorer les performances, ce choix sera proposé dans le cadre d'une autre thèse de recherche [Liu to appear].

Comme pour *EXODUS*, notre stratégie de recherche sera orientée par l'attribution d'un coefficient d'amélioration à chaque règle de transformation. Toutefois, le choix de la valeur du coefficient ne peut être attribué de la même manière que pour *EXODUS*. En effet, dans le cadre de la médiation, le coefficient d'amélioration des règles de transformation peut varier en fonction du coût d'évaluation des requêtes sur les sources distantes ou entre deux sources distinctes. Seul le modèle de coût peut nous permettre de donner ces détails. Ainsi, le coefficient d'amélioration sera calibré puis raffiné par historique pour lui permettre de rester à jour. Nous pouvons définir le coefficient d'amélioration grâce au modèle de coût grâce à la formule suivante :

$$coeff = \frac{cost(\varphi(tgv))}{cost(tgv)}$$

Cette formule définit le coefficient par le rapport entre le coût d'évaluation théorique

du TGV modifié sur le coût d'évaluation théorique du TGV initial.

Toutefois, le modèle de coût seul ne suffit pas pour définir le coefficient d'amélioration d'une règle de transformation. En effet, les catégorisations des règles définies précédemment influent sur le coût d'évaluation du TGV. Chaque catégorie apporte une information particulière dépendant de ses propres caractéristiques. Concernant les règles de transformations logiques, celles-ci ne tiennent pas compte des informations présentes sur les sources distantes. Alors que les règles de transformations physiques tiennent compte des informations provenant des sources.

Ainsi, cette différence, entre les transformations logiques et physiques, influe sur le coefficient d'amélioration puisque les transformations logiques préparent le TGV à l'évaluation physique. Ainsi, le coefficient des transformations logiques doit être plus intéressant que les transformations physiques, celui-ci sera donc affecté d'un facteur d'influence de $\times 0.5$ pour représenter cette importance. Concernant les transformations utilisateurs, celles-ci peuvent aussi bien influencer les transformations logiques et physiques, changer le coût d'évaluation (exemple de la fonction *contains*), il n'est donc pas possible de déterminer de manière automatique son influence sur l'optimisation. Ainsi, son facteur d'influence sera déterminé par l'utilisateur qui pourra déterminer l'importance de la transformation.

Ainsi, grâce à un coefficient d'amélioration défini pour chaque règle de transformation de l'optimiseur, nous pouvons orienter la stratégie de recherche. Ce coefficient défini par le rapport du coût d'évaluation théorique après transformation sur le coût après transformation est raffiné par un historique du coût d'évaluation fait sur les sources. De plus, un facteur d'influence est affecté à chaque règle de transformation définie en fonction de sa catégorisation permettant de donner une plus grande importance aux transformations logiques et transformations utilisateurs spécifiques.

4.4.3 Conclusion

La stratégie de recherche définie dans l'optimiseur s'appuie sur un modèle de coût défini par calibrage des sources avec raffinage sur historique. Cette stratégie incrémentale définie un cadre d'optimisation pour effectuer des tests de performances (section 5.5). Ue stratégie plus efficace sera définie dans la thèse de [Liu to appear], plus adaptée au modèle de coût permettant d'annoter des estimations sur les TGV. L'annotation des TGV permet de donner une estimation du coût d'évaluation sur des ensembles d'éléments présents dans le TGV. Le parcours de l'espace de recherche des plans est orienté grâce à l'attribution d'un coefficient d'amélioration des règles de transformations. Ce coefficient est orienté lui-même par calibrage de celles-ci grâce au modèle de coût avec raffinage par historique. Ces coefficients sont influencés par un facteur déterminé grâce à la catégorisation des règles de transformation.

4.5 Conclusion

L'optimiseur basé sur le modèle TGV propose un ensemble d'outils permettant de manipuler les TGV : une algèbre abstraite, une base d'annotation, un langage de définition de règles de transformation, une catégorisation des transformations, un modèle de coût générique et une stratégie de recherche.

Une algèbre abstraite est proposée pour définir une évaluation des TGV de manière intuitive indépendante d'une algèbre physique. Celle-ci repose sur une décomposition des ensembles caractéristiques des TGV pour donner un ensemble d'opérateurs fonctionnels capables d'évaluer des documents XML grâce à notre modèle de représentation.

Une base d'annotation générique est associée au modèle pour représenter n'importe quelle information que nous souhaitons intégrer à la représentation. Ces annotations regroupent des ensembles d'éléments de TGV pour leur associer une annotation quelconque. Ainsi, il est possible de définir des annotations de méta-données, capacités fonctionnelles, coût d'évaluation, choix d'algorithmes flexibles [Calmès *et al.* 2003]... Un TGV qui a été annoté est appelé TGV physique. Ce TGV annoté peut alors être optimisé.

L'optimiseur modifie un TGV à l'aide de règles de transformation. Ces règles de transformation peuvent être intégrées dans le système grâce à un langage de définition de règles de transformation. Ce langage repose sur les opérations des types abstraits de données définies par le modèle, permettant ainsi de spécifier librement toute règle possible. Grâce à ce langage de règle, l'optimiseur est *extensible*, car l'ensemble des règles de transformation peut être modifié.

Chaque règle de transformation déclarée dans l'optimiseur est classée dans une catégorie. Trois classes sont définies. Les transformations logiques équivalentes transforment un TGV à l'aide des simples connaissances des TGV sans modifier le résultat de l'évaluation. Les transformations physiques équivalentes transforment un TGV à l'aide des informations présentent dans le TGV et dans les annotations tout en préservant le résultat de l'évaluation. Les transformations utilisateurs sont définies en fonction du contexte dans lequel s'intègre l'optimiseur. Il est parfois utile de ne pas préserver l'équivalence pour répondre de manière plus «correcte» à une requête. L'exemple de règles de transformations sémantiques permet de répondre à une requête en fonction du thème choisi, et non du schéma exacte des données (qui peut être bien différent). Cette catégorisation des règles de transformation permet d'orienter la stratégie de recherche de l'optimiseur.

Pour permettre à l'optimiseur d'améliorer les TGV à l'aide des règles de transformation sans générer l'ensemble des TGV possibles, une stratégie de recherche a été définie ; elle permet donc de réduire l'espace de recherche. Pour cela, un modèle de coût est nécessaire pour déterminer le coût d'évaluation de chaque TGV, celui-ci est défini par calibrage des sources et raffiné par historique. Chaque ensemble d'élé-

ments d'un TGV est annoté par son coût d'évaluation, le coût d'évaluation du TGV est alors défini par un système de formules. La stratégie de recherche s'appuie sur ce modèle de coût pour définir un coefficient d'amélioration pour chaque règle de transformation. Ce coefficient défini entre 0 et 1, est fixé par calibrage des règles et par raffinage par historique. De plus, le coefficient d'amélioration est orienté par un facteur d'influence réglé par la catégorisation de la règle de transformation. Au final, l'optimiseur choisi la règle applicable ayant le meilleur coefficient d'amélioration.

Ainsi, nous avons proposé un optimiseur extensible basé sur le modèle des TGV capable de définir son évaluation, son coût et ses transformations. L'optimiseur détermine ainsi un TGV quasi-optimal qui peut être évalué par le médiateur.

Une partie des techniques décrites dans ce chapitre a été mise en œuvre dans le cadre du médiateur *XLive*. La décomposition des TGV ainsi que les annotations permettent de générer les plans logiques et physiques dans le médiateur. Un modèle de coût générique a été intégré dans les annotations pour permettre à l'optimiseur de donner un coût théorique à chaque plan. Pour le moment, les règles de transformation sont implémentées dans l'optimiseur ; le parseur spécifique au langage de règle de transformation est en cours d'implémentation.

Chapitre 5

XLive : un système de médiation

Le système de médiation présenté dans cette thèse résulte de l'évolution du système de médiation *XMLMediator* de *e-XMLMedia* [Gardarin *et al.* 2002] et de l'intégration des différentes techniques que nous avons pu aborder dans cette thèse. *XMLMediator* a été développé au sein de la société *e-XMLMedia*, les bases de cette architecture ont été utilisées pour développer le système de médiation **XLive** au laboratoire *PRiSM*.

L'objectif du projet *XLive* est de pouvoir fédérer des sources de données hétérogènes et distribuées, tout en utilisant les technologies "XML". Ainsi, *XLive* est une architecture de médiation "tout-XML" permettant à un utilisateur d'interroger le médiateur dans le langage *XQuery* et de récupérer un résultat sous forme d'un document XML. Les requêtes sont représentées en interne par les *TGV*. L'évaluation des données XML est facilitée par l'utilisation d'une *XAlgebre* (appelée aussi *XAlgebra*). Des adaptateurs permettent la communication avec toutes sortes de sources de données : natives XML (Xyleme, eXist, Xhive), relationnelles (Oracle, MySQL, Microsoft Access), fichiers XML, Web Services (Google API, Amazon).

Dans ce chapitre, nous présentons le fonctionnement du système *XLive* avec son architecture, les adaptateurs et la *XAlgebre* (section 5.1). Nous proposons ensuite une étude qualitative des TGV grâce aux cas d'usage du W3C (section 5.2). Le médiateur *XLive* est alors testé sur le cas d'usage XMP dont les performances sont développées dans la section 5.3. Puis, une description de l'optimiseur extensible est proposée dans la section 5.4. Enfin, une étude quantitative à l'aide de bancs d'essai [Dragan et Gardarin 2005] valident nos travaux (section 5.5).

5.1 Architecture de Médiation

XLive est un système de médiation "tout-XML" implémenté en **Java** permettant de fédérer des sources de données hétérogènes et distribuées. L'architecture de médiation DARPA I3 a été retenue. Elle permet dans notre contexte de traiter les requêtes XQuery pour produire un résultat pouvant intégrer des données extraites par l'ensemble des sources reliées au médiateur. La XAlgèbre permet de traiter les données sous forme de flux XML provenant des différentes sources intégrées pour construire un résultat final. Les adaptateurs dialoguent avec les sources et permettent au médiateur de connaître les informations nécessaires pour l'évaluation et l'optimisation des requêtes.

Nous allons tout d'abord étudier l'architecture de *XLive* (section 5.1.1), puis nous détaillons les opérateurs de la *XAlgèbre* (section 5.1.2).

5.1.1 Composants de XLive

L'architecture de médiation *XLive* est décomposée en une architecture trois tiers comme nous pouvons le voir dans la figure 5.1 :
- une interface de communication permet de transmettre les informations entre l'utilisateur et le médiateur ;
- le médiateur traite la requête de l'utilisateur, puis les données envoyées par les sources ;
- des adaptateurs traduisent les requêtes et les données entre le médiateur et les sources ciblées.

Le cœur de cette architecture se situe dans le médiateur. Il est décomposé en modules reliés entre eux permettant l'évaluation des requêtes. Les différents modules le composant ont les fonctions suivantes :

1. l'**analyseur** permet de canoniser une requête XQuery et de la transformer dans le modèle interne : les *TGV* ;
2. l'**optimiseur** de requêtes transforme les *TGV* logiques et physiques pour obtenir une représentation optimale ;
3. le **gestionnaire d'adaptateurs** permet le dialogue entre les adaptateurs (couche inférieure) et le médiateur (couche centrale) ;
4. le **générateur de plans** permet la transformation d'un *TGV logique* en un *TGV physique* par l'intermédiaire des annotations (algorithmes, coûts, choix des sources...) ;
5. l'**évaluateur de requêtes** exécute l'arbre algébrique.

L'**analyseur** canonise les requêtes XQuery à l'aide des règles de canonisation (sections 2.2.3 et 3.2). La requête canonisée est alors traduite dans le modèle de représentation de requêtes : les *TGV*. La grammaire XQuery a été intégrée au parseur *JavaCC* qui reconnaît les mots clés définis dans la grammaire pour les transformer

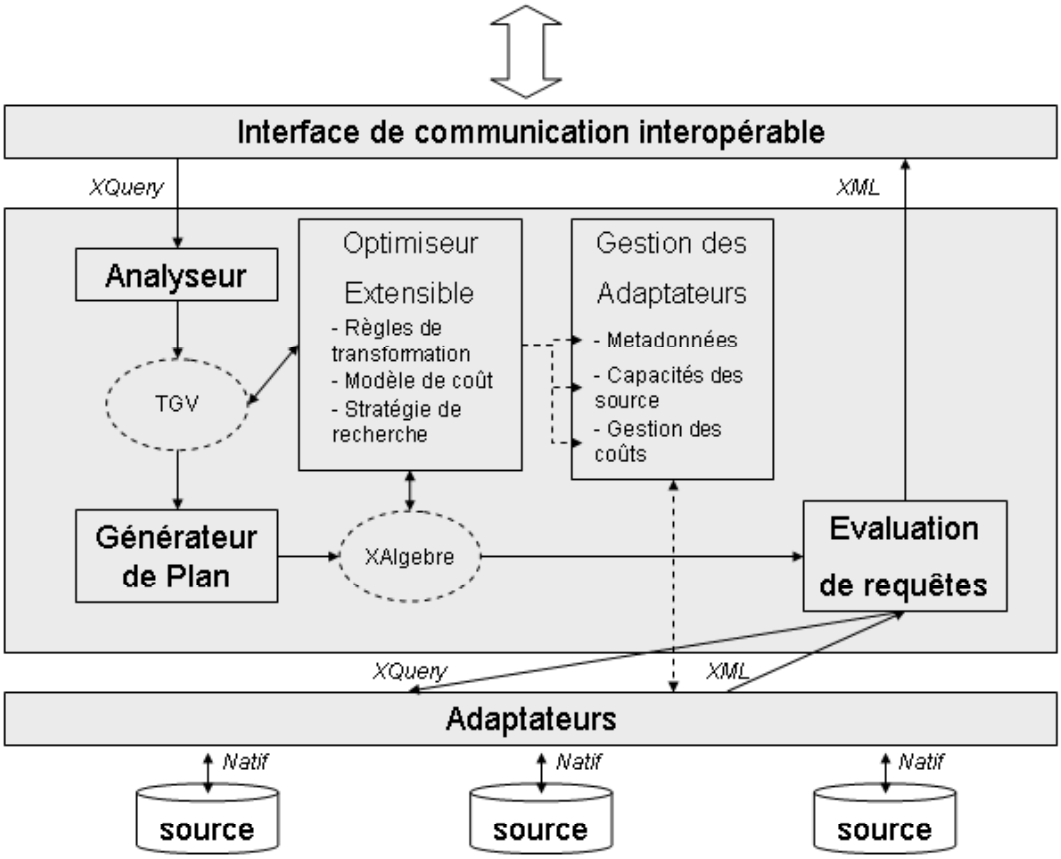


FIG. 5.1 – Architecture de médiation

en objets *Java* du modèle TGV. L'implémentation du parseur permet d'effectuer simultanément la canonisation et la traduction.

L'**optimiseur** de requêtes manipule les *TGV* à l'aide des règles de transformation définies par le langage de règles. Un parseur *JavaCC* reconnaît les mots clés du langage pour créer des règles d'optimisation sous forme de classes. Chaque classe possède une méthode de vérification des conditions, et une méthode de transformation ; de plus le coefficient d'amélioration lui est associé pour orienter l'optimiseur. La base d'annotations est implémentée sous forme d'objets, elle permet de lister les composants du TGV pour fournir des annotations. La stratégie de recherche manipule les annotations et les règles pour transformer un TGV et obtenir au final un TGV optimal.

Le **gestionnaire d'adaptateurs** permet le dialogue entre les adaptateurs (couche inférieure) et le médiateur (couche centrale). Il récupère les informations que les sources publient : métadonnées, capacités, modèle de coûts. Il permet au modèle

de coût de l'optimiseur d'avoir les informations nécessaires à l'annotation des TGV. Les adaptateurs récupèrent les données sous forme de flux *SAX* (parseur de données XML) qui seront alors transformées dans le modèle interne à la *XAlgèbre*.

Le **générateur de plans** transforme un *TGV logique* en un *TGV physique* par l'intermédiaire des annotations (algorithmes, coûts, choix des sources...). En effet, l'optimiseur opère d'abord sur le TGV logique, puis sur le TGV physique. Les algorithmes des opérateurs et les choix d'évaluation (distant ou local) sont annotés sur le TGV. Une fois l'ensemble des traitements effectués, le TGV physique est alors traduit en opérateurs de la *XAlgebre* formant alors un arbre algébrique d'évaluation.

L'**évaluation de requêtes** permet l'évaluation d'opérateurs de la *XAlgebre*. L'arbre algébrique est évalué en flux XML, permettant de produire au fur et à mesure des résultats en XML. Les adaptateurs sont reliés aux feuilles de l'arbre algébrique, et l'interface de communication à la racine.

L'architecture *XLive* permet donc l'exécution de requêtes XQuery (non typées). L'ensemble de ses composants manipule des requêtes XQuery et des documents XML par l'intermédiaire des TGV et de la XAlgebre. Nous allons maintenant détailler cette algèbre interne au médiateur, permettant l'évaluation des documents XML.

5.1.2 XAlgèbre

La *XAlgèbre* [Dang-Ngoc 2003] est une algèbre d'évaluation de requêtes sur documents XML. C'est une algèbre adaptée de l'algèbre relationnelle pour traiter des documents XML. Chaque opérateur (appelé *XOperateur*) de la XAlgèbre manipule des relations (appelées *XRelations*) contenant des tuples (appelés *XTuples*). Comme pour l'algèbre relationnelle, les XOpérateurs de la XAlgèbre forment un arbre algébrique d'évaluation manipulant des XRelations.

Les XTuples sont une représentation des données XML. Ils comportent deux parties distinctes. Une partie contenant les données XML, codé sous forme d'arbre DOM, l'autre partie est un tableau contenant des attributs pointant vers les éléments utiles contenus dans les données. Les attributs sont utilisés en paramètres des XOpérateurs pour leur permettre de récupérer la valeur requise directement. Cette référence permet d'évaluer un XTuple plus rapidement, car il n'a pas à chercher la valeur demandée. La figure 5.2 illustre un exemple de XRelation contenant des XTuples. La partie de gauche représente les références vers les valeurs contenues dans les arbres DOM (partie de droite). Ces arbres ont pour racine le nom de variable de provenance de la requête ($x et $y).

Les XTuples sont créés lorsque les données sont récupérées des sources de données distantes. Les éléments du document XML sont construits en arbre DOM au fur et à mesure. Lorsqu'un chemin (suite d'éléments) demandé est présent dans l'arbre, il est alors ajouté au tableau de références. Cette construction n'a qu'un faible surcoût de construction puisqu'il est fait à la volée, en même temps que la construction de

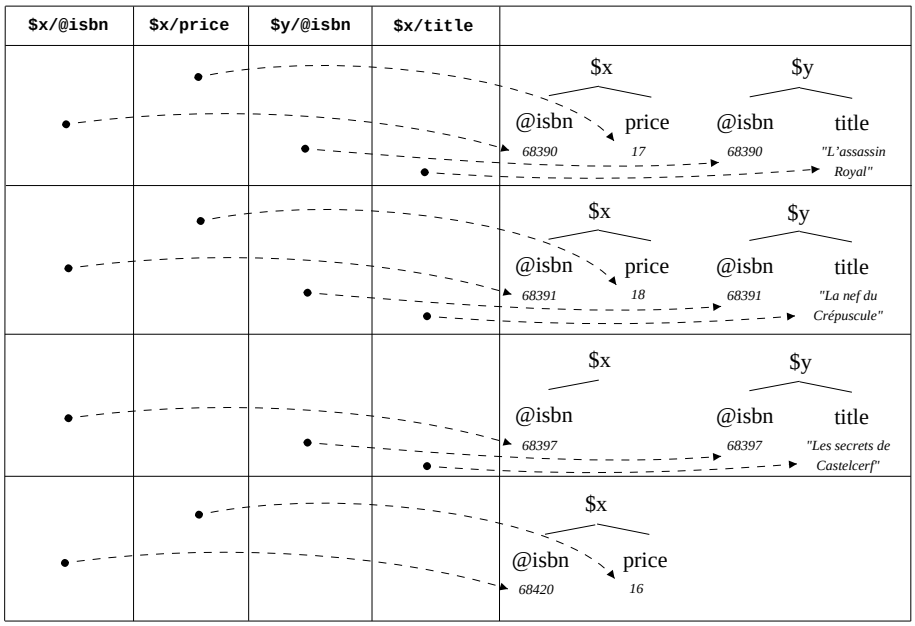


FIG. 5.2 – Exemple de XTuples

l'arbre qui est obligatoire.

Chaque opérateur de la XAlgèbre prend une ou plusieurs XRelation en entrée, et traite XTuple par XTuple les informations qu'elles contiennent. Afin de traiter l'intégralité des requêtes supportées par le médiateur, un ensemble de XOpérateurs sont disponible dans cette algèbre :

- **XSource** : Permet la connexion vers un adaptateur auquel est associé une requête atomique (spécifique pour la source). Cet opérateur permet la construction des XTuples utiles dans les autres XOpérateurs ;
- **XJoin** : Permet de relier deux XRelations puis de fusionner les résultats selon un prédicat de jointure. Les arbres DOM et les références sont alors mises à jour dans chaque XTuple. Plusieurs algorithmes sont supportés par l'opérateur, dépendant des choix de l'optimiseur de requêtes (boucle imbriquée, hachage, double hachage). Il permet aussi l'imbrication des résultats par une agrégation des arbres joints ;
- **XConstraint** : Permet de restreindre les XTuples par application de prédicats. Le chemin utilisé par le prédicat est sélectionné dans les références et vérifie la valeur correspondante dans l'arbre DOM. Cet opérateur se révèle indispensable lorsqu'une source n'a pas les capacités de traitement nécessaire à l'évaluation de la requête (la contrainte ne peut alors être déléguée à la source) ;
- **XOrderBy** : Permet d'ordonnancer les XTuples par valeurs donnée par le contenu de la référence correspondante ;
- **XUnion, XInter, XDiff** : Permet d'appliquer une opération ensembliste entre deux XRelations. XUnion est très utilisée dans le cas où deux sources répondent à la même requête (besoin de fusion des résultats) ;
- **XMin, XMax, XCount, XAvg, XSum, etc...** : Permet d'appliquer une fonc-

tion d'agrégation sur une XRelation. Le chemin donné en paramètre donne la référence requise pour cet agrégat, la valeur correspondante est alors utilisée pour l'agrégat ;
- **XIf** : Permet d'appliquer une expression conditionnelle (If/Then/Else) sur une XRelation. Le résultat de la contrainte donnée en paramètre donne le résultat provenant de la XRelation ;
- **XConstruct** : Permet de construire un document XML à partir d'un XTuple (références requises) et des balises de construction. Cet opérateur peut produire une chaîne de caractères (affichage des résultats), un nouvel XTuple (résultat temporaire pour réutilisation), voir des événements SAX (utilisation externes au médiateur).

La XAlgèbre permet donc de produire un résultat à partir d'une composition d'XOpérateurs. Afin de définir une transformation des TGV physiques dans un plan composé d'XOpérateurs, une correspondance entre l'algèbre abstraite et la XAlgèbre est nécessaire. Grâce à cette correspondance, chaque requête peut être traduite en opérateurs de la XAlgèbre, et l'arbre algébrique résultant être évalué. Le tableau 5.1 donne les correspondances entre les éléments des TGV (colonne de gauche), les opérateurs de l'algèbre abstraite (colonne centrale) et les opérateurs correspondant dans la XAlgèbre (colonne de droite). Pour chaque opérateur de la XAlgèbre, les références sont données par les motifs d'arbre de l'algèbre abstraite, et les arbres DOM par l'ensemble d'entrée de l'opérateur précédent : τ.

TGV	**Eval**	**XAlgebre**
STP	$\phi(stp, \tau)$	XSource
ITP	$\phi(itp, \pi(itp, \tau))$	XSource
RTP + links	$\beta(rtp, \pi(rtp, \tau))$	XConstruct
ATP	$\alpha(atp, \tau)$ $\alpha(atp, \tau)$	XMin, ... XOrderBy
Constraint	$\sigma(rtp, \tau_r)$	XConstraint
JoinHyperlink	$\sigma(rtp, \tau_r)$	XJoin
SetHyperlink	$\sigma(rtp, \tau_r)$	XUnion, ...
IfThenElseHyperlink	$\sigma(rtp, \tau_r)$	XIf

Tab. 5.1 – Correspondance algèbre abstraite TGV et XAlgebre

Comme nous pouvons le voir dans le tableau 5.1, les éléments composant les TGV sont transformés en XOpérateurs de la XAlgèbre :
- **STP** : le motif d'arbre source donne la requête à produire au niveau de la source, ainsi que les éléments à filtrer (ϕ), donnant les éléments nécessaires à l'XOpérateur XSource ;
- **ITP** : le motif d'arbre intermédiaire composé avec le motif d'arbre source (ϕ et π) permet de produire de la même manière un XSource capable de filtrer les résultats provenant des sources ;
- **RTP** : le motif d'arbre résultat contient les informations nécessaires pour l'opérateur XConstruct. Le motif d'arbre donne le document à construire (β) et les hyperliens les références à projeter (π) ;

- **ATP** : le motif d'arbre d'agrégation permet de gérer les fonctions d'agrégats (α). Le motif d'arbre lié à un hyperlien de projection donne le chemin nécessaire à l'agrégation des données. Pour l'opérateur $XOrderBy$, la règle de canonisation de la clause *order by* (section 3.2.1) donne une fonction d'agrégation qui est contenue dans un ATP ; il correspond alors à cet XOpérateur ;
- **Constraint** : les contraintes génèrent un XOpérateur capable de filtrer (σ) les XTuples. Les chemins nécessaires à la restriction sont obtenus par les *Constraint-Link*, reliés à un nœud. Les formules booléennes sont obtenues grâce aux hyperliens de contrainte ;
- **JoinHyperlink** : un hyperlien de jointure génère un XJoin dont les deux XRelation proviennent des motifs d'arbres liés, et la contrainte permet de joindre (σ) les deux ensembles ;
- **SetHyperlink** : un hyperlien ensembliste génère un XOpérateur ensembliste (XUnion, XInter ou XDiff) reliant les deux XRelation provenant des motifs d'arbre reliés par l'hyperlien ;
- **IfThenElseHyperlink** : un hyperlien conditionnel permet de générer un XIf qui prend une XRelation provenant de l'ensemble des motifs d'arbre reliés, et donne un résultat dépendant de la contrainte associée.

Les éléments composant les TGV sont traduits en XOpérateurs de la XAlgèbre. La composition d'un arbre algébrique est induite par la composition des opérateurs de l'algèbre abstraite, provenant elle-même de la traduction d'un TGV. Ainsi, nous pouvons traduire une requête modélisée par les TGV en un arbre algébrique qui pourra alors être évalué.

5.2 Traitements de Requêtes

Les processus d'analyse de requête, canonisation et traduction ont été implémenté en *Java*. Le parseur de requête *JavaCC* permet de reconnaître les mots clés et ainsi de vérifier la grammaire et la syntaxe du langage XQuery. La canonisation de chaque élément est appliquée juste avant la traduction de la requête XQuery en TGV, qui se fait à la volée. Ainsi, nous pouvons obtenir une instance de TGV en combinant les deux étapes dans le même parseur de requêtes.

La base de test de notre parseur est constitué des requêtes fournies par le W3C : les cas d'usage (ou *use-cases*) [W3C 2006b]. Les tableaux présentés ci-dessous résument le contenu de chacune des requêtes des cas d'usage. La colonne de droite précise si la requête est représentable à l'aide des TGV (canonisation + traduction + représentation graphique). Nous pouvons distinguer neuf cas d'usage différents, correspondant à des caractéristiques particulières de XQuery et des documents XML.

- **XMP** : Expérimentations et exemples de requêtes types pour XQuery ;
- **TREE** : Requêtes de préservation de l'arborescence des éléments ;
- **SEQ** : Requêtes séquences sur les éléments ;
- **R** : Requêtes sur des données relationnelles (stockées en XML) ;

- **SGML** : Requêtes tirées du cas d'usage de SGML (1992), transcrites en XQuery ;
- **STRING** : Requête de recherche de chaîne de caractères dans un document XML ;
- **NS** : Requête sur l'utilisation des namespace, fonctions ou éléments ;
- **PARTS** : Définition de fonctions fortement récursives ;
- **STRONG** : Requête de typage fort sur des données typées (fonctions, instance, éléments, typeswitch).

Pour ne pas surcharger ce rapport de thèse, nous ne représentons pas ici toutes les requêtes des cas d'usage canonisées ou sous forme de TGV. Toutefois, pour illustrer ces cas d'usage, nous proposons en annexe F, le cas d'usage XMP, les douze requêtes sont représentées sous forme de TGV. Concernant les autres cas d'usage, la requête la plus caractéristique du lot de requêtes de chaque cas d'usage est proposée dans cette annexe.

Les tableaux suivants listent chacun des cas d'usage pour chaque catégorie proposé par le W3C. La colonne "description" donne une courte énumération du contenu des requêtes du cas d'usage. La colonne TGV indique si le cas d'usage est modélisable par un TGV à l'heure actuelle.

Nom	Description	TGV
Q1	Domaine simple, double prédicat, projection d'un élément et d'un attribut	OK
Q2	Triple définition de domaine, projection d'éléments	OK
Q3	Définition de domaine simple	OK
Q4	Let, fonctions, filtres, ordonnancement, quantificateur 'some', imbrication de requête	OK
Q5	Double domaine de définition, jointure	OK
Q6	Fonction d'agrégation, imbrication avec filtre, opération conditionnelle	OK
Q7	Double prédicat, ordonnancement	OK
Q8	let, redéfinition de domaine, filtre complexe, élément '*' dans un XPath, fonction	OK
Q9	Opération ensembliste sur domaine, fonction "contains"	OK
Q10	Let, redéfinition de domaine, filtre, fonction d'agrégat	OK
Q11	Deux requêtes imbriquées	OK
Q12	Double définition de domaine, let, ordonnancement, prédicat de jointure, composition de contraintes	OK

Tab. 5.2 – Cas d'usage XMP : Expériences et exemples

Le tableau 5.2 illustre le cas d'usage XMP, correspondant aux requêtes d'expérimentations et d'exemples de requêtes types. Celles-ci se caractérisent par les combinaisons des particularités du langage *XQuery* : (prédicats, filtres, quantificateurs, imbrication, jointure, agrégation, opérations ensemblistes et conditionnelles, composition de contraintes). Les TGV générés par ce cas d'usage contiennent de nombreux motifs d'arbre dû à la présence des filtres, des fonctions d'agrégation et des nombreuses définitions de domaines. Chaque requête associée à une représentation est disponible en annexe F.1.

Nom	Description	TGV
Q1	Définition de fonction récursive, redéfinition de domaine, attribut quelconque '@*'	OK
Q2	Définition de domaine simple, attribut quelconque '@*'	OK
Q3	Deux fonctions d'agrégation	OK
Q4	fonction d'agrégation	OK
Q5	let, redéfinition de domaine, fonction d'agrégation	OK
Q6	Définition de fonction récursive, fonction d'agrégation, attribut quelconque '@*'	OK

Tab. 5.3 – Cas d'usage TREE : préservation de la hiérarchie

Le tableau 5.3 illustre le cas d'usage TREE, correspondant aux requêtes permettant de préserver l'arborescence des éléments dans un document XML. Les requêtes sont caractérisées par des appels de fonction et des fonctions d'agrégations. Une représentation de la requête 6 (la plus représentative) est proposée en annexe F.2.

Nom	Description	TGV
Q1	Filtre, séquence	OK
Q2	Filtres, séquence	OK
Q3	Let, double séquence, double filtres, prédicat de jointure	OK
Q4	Filtres, séquences, composition de contraintes, quantificateur 'some', prédicat de jointure	OK
Q5	Déclaration de fonction, composition de contraintes, opération ensembliste, prédicat de jointure, let, filtres, séquences	OK

Tab. 5.4 – Cas d'usage SEQ : Séquence

Le tableau 5.4 illustre le cas d'usage SEQ, correspondant aux requêtes de séquences sur les éléments d'un document XML. Elles sont caractérisées par l'apparition de parenthèses dans les expressions de chemin (séquences) et une utilisation de filtres complexes. Les TGV générés définissent des motifs d'arbre reliés les uns aux autres par des hyperliens de spécialisation et de généralisation, dû à la canonisation des requêtes. Une représentation de la requête 4 (la plus représentative) est proposée en annexe F.3.

Le tableau 5.5 illustre le cas d'usage R, correspondant aux requêtes sur des données relationnelles, stockées au format XML. Les requêtes sont caractérisées par de nombreuses jointures entre les domaines, les clauses "*where*" contiennent de nombreux prédicats. De plus, les fonctions d'agrégations et d'ordonnancement sont fréquemment utilisées. Les TGV donnent pour la plupart des motifs d'arbre "plats", contenant beaucoup d'éléments sous la racine. Des groupes de motifs d'arbre sont reliés aux motifs d'arbre d'agrégation formant des ensembles d'évaluation bien distincts. Une représentation de la requête 14 (la plus représentative) est proposée en annexe F.4.

Nom	Description	TGV
Q1	Prédicats, fonctions, prédicats de jointure, ordonnancement	OK
Q2	Let, filtre, prédicat de jointure, fonction, ordonnancement, fonction d'agrégation	OK
Q3	prédicats, prédicat de jointure	OK
Q4	Composition de contraintes, Filtre, prédicat de jointure	OK
Q5	Prédicats, prédicats de jointure, composition de fonctions, fonction d'agrégation, filtres, ordonnancement	OK
Q6	Let, filtre, prédicat de jointure, composition de contraintes, fonction d'agrégation, prédicat	OK
Q7	filtres, opération booléenne, let, prédicat de jointure	OK
Q8	let, filtre, fonctions, fonction d'agrégation	OK
Q9	let, fonction, imbrication de requête, filtre, prédicats, fonction d'agrégation	OK
Q10	prédicats de jointures, fonction d'agrégation, requête imbriquée, filtre, ordonnancement	OK
Q11	let, imbrication de requête, filtre, prédicats de jointure	OK
Q12	définition de fonction, fonction, let, filtre, prédicats de jointure, fonction d'agrégation	OK
Q13	filtres, prédicats de jointure, let, ordonnancement, fonctions d'agrégation	OK
Q14	fonction, imbrication de requête, filtre, prédicat de jointure, fonctions d'agrégation, composition de contrainte, ordonnancement	OK
Q15	let, filtre, prédicat de jointure, fonction d'agrégation, composition de contrainte	OK
Q16	let, filtre, prédicat de jointure, ordonnancement, opération conditionnelle	OK
Q17	quantificateur 'every', imbrication de requête, prédicats de jointure	OK
Q18	ordonnancements, imbrications de requête, filtre, prédicat de jointure, let	OK

Tab. 5.5 – Cas d'usage R : Relational Data

Nom	Description	TGV
Q1	liens ancêtre/descendant	OK
Q2	lien ancêtre/descendant, lien parent/enfant	OK
Q3	lien ancêtre/descendant, fonction	OK
Q4	liens ancêtre/descendant, séquences, filtres	OK
Q5	lien ancêtre/descendant, filtre	OK
Q6	lien ancêtre/descendant	OK
Q7	lien ancêtre/descendant, composition de fonctions	OK
Q8	liens ancêtre/descendant, filtres, fonction	OK
Q9	liens ancêtre/descendant, imbrication de requête, filtre, prédicat de jointure	OK
Q10	liens ancêtre/descendant, let, filtres, prédicat de jointure, fonction	OK

Tab. 5.6 – Cas d'usage SGML : Standard Generalized Markup Language

Le tableau 5.6 illustre le cas d'usage SGML, correspondant aux requêtes reprises du cas d'usage SGML (1992), celles-ci ont été transcites en *XQuery*. Les requêtes se caractérisent par l'utilisation des relations d'ascendance entre les éléments (pa-

rent/enfant et ancêtre/descendant). Des caractéristiques du langage *XQuery* sont aussi utilisées : filtres, jointure, séquences. Les TGV se représentent principalement par un motif d'arbre source unique sur un motif d'arbre source, associé à des contraintes. Une représentation de la requête 8 (la plus représentative) est proposée en annexe F.5.

Nom	Description	TGV
Q1	lien ancêtre/descendant, filtre, fonction 'contains'	OK
Q2	Déclaration de fonction, let, filtre, prédicat de jointure, quantificateurs 'some', fonction 'contains', opérations booléennes complexes, composition de fonctions	OK
Q3	*A été supprimée*	
Q4	Déclaration de fonction, let, filtre, prédicats de jointure, fonction 'contains', quantificateur 'some'	OK
Q5	Fonction 'contains', fonctions, séquence, filtre	OK

Tab. 5.7 – Cas d'usage STRING : recherche de chaînes de caractère

Le tableau 5.7 illustre le cas d'usage STRING, correspondant aux requêtes de recherche textuelle dans les documents XML. Celles-ci se caractérisent par l'utilisation de la fonction '*contains*' de recherche de chaîne de caractères, mais aussi des quantificateurs, des filtres et des jointures. Les TGV sont composés de plusieurs domaines, reliés à des contraintes reliées par des hyperliens de contraintes complexes, les motifs d'arbre d'agrégation sont présents dûs à la canonisation des quantificateurs. Une représentation de la requête 2 (la plus représentative) est proposée en annexe F.6.

Nom	Description	TGV
Q1	Imbrication de requête, opération ensembliste, élément et attribut quelconque '*' et '@*', fonction 'namespace-uri'	OK
Q2	Élément avec namespace	OK
Q3	Element quelconque, filtre, attribut quelconque avec namespace	OK
Q4	Lien ancêtre/descendant, attribut avec namespace	OK
Q5	Lien ancêtre/descendant, élément et attribut avec namespace, filtre, prédicat	OK
Q6	Lien ancêtre/descendant, éléments et attributs avec namespace, filtre	OK
Q7	Let, namespace, fonction 'namespace-uri', prédicat de jointure, namespace quelconque '* :'	OK
Q8	Lien ancêtre/descendant, opération ensembliste, namespace, namespace quelconque '* :', prédicat	OK

Tab. 5.8 – Cas d'usage NS : Namespaces

Le tableau 5.8 illustre le cas d'usage NS, correspondant aux requêtes utilisant les espaces de noms dans les fonctions ou les éléments. Les requêtes se caractérisent par l'utilisation des namespace «xxx :» devant les noms et les fonctions. Certains éléments sont déclarés avec le symbole «*». De nombreux attributs sont présents,

associés à des filtres, ainsi que la fonction «*namespace-uri*». Les TGV sont simples, contenant des nœuds dans lesquelles apparaissent les espaces de noms. Une représentation de la requête 7 (la plus représentative) est proposée en annexe F.7.

Nom	Description	TGV
Q1	Définition de fonction récursive, requête imbriquée, prédicat de jointure, lien ancêtre/descendant, filtre, fonction	OK

Tab. 5.9 – Cas d'usage PARTS : Fonction récursive

Le tableau 5.9 illustre le cas d'usage PARTS, contenant une requête de déclaration de fonction fortement imbriquée. Celle-ci contient par ailleurs un prédicat de jointure, de filtres, des définitions de domaines récursifs, des axes de chemins ancêtre/descendant et des imbrications de requêtes. Le TGV résultant ne forme qu'un ensemble de motifs d'arbre définissant la fonction avec les éléments d'entrées (variables) et le motif d'arbre résultat en sortie. La requête du cas d'usage PARTS est proposée en annexe F.8.

Nom	Description	TGV
Q1	Lien ancêtre/descendant, filtre, instance, namespace	NON
Q2	Déclaration de fonction, quantificateur 'some', imbrication de requête, namespace, element typé	NON
Q3	Déclaration de fonction, quantificateur 'some', imbrication de requête, namespace, fonction, element typé	NON
Q4	Déclaration de fonction, 'typeswitch', namespace, élément typé, let, fonctions	NON
Q5	Déclaration de fonction, éléments typés, prédicat de jointure, let	NON
Q6	let, lien ancêtre/descendant, élément typé, opération conditionnelle, instance, quantificateur 'every'	NON
Q7	Déclaration de fonction, élément typé, fonction, prédicats	NON
Q8	Eléments typés, prédicats, fonction, lien ancêtre/descendant	NON
Q9	Déclaration de fonction, éléments typés, prédicats, fonction, requête imbriquée	NON
Q10	Déclaration de fonction, éléments typés, prédicat de jointure, imbrication de requête	NON
Q11	Déclaration de fonction, élément typé, let, prédicat de jointure, fonction d'agrégation	NON
Q12	let, éléments typés, filtres, prédicats de jointure, fonctions, ordonnancement, requêtes imbriquées, fonction d'agrégation, composition de contrainte	NON

Tab. 5.10 – Cas d'usage STRONG : Exploitation de typage fort de données

Le tableau 5.10 illustre le cas d'usage STRONG, correspondant aux requêtes de typages fort, s'appuyant sur des données fortement typées. Les requêtes se caractérisent par l'utilisation des opérateurs '*instance of*', '*typeswitch*', '*element(...)*' et '*as*'. Ces opérateurs n'ayant pas été intégrés au modèle TGV, ceux-ci ne peuvent

avoir de correspondance. Toutes les autres opérations contenues dans les requêtes sont toutefois reconnues par notre modèle.

Ainsi, sur l'ensemble des neuf cas d'usage proposés par le W3C, seulement huit peuvent être canonisées et représentées. Compte tenu du contexte de médiation, le typage contenu dans le cas d'usage STRONG ne peut être reconnu. En effet, il est difficile dans le contexte de médiation de tenir compte des données typées à partir du moment où les sources ne publient pas toujours ce genre d'information. De plus, le surcoût de vérification des types de données pour chaque source n'est pas négligeable.

Toutefois, il serait possible de prendre en compte le typage dans notre modèle. Les opérations '*instanceof*', '*as*' et '*element(...)*' impliquant un typage des éléments, peuvent être traitées comment des contraintes sur des nœuds. Ainsi, la plupart des cas d'usage de typage pourront être reconnus.

Par contre, l'opération '*typeswitch*' pose un problème de typologie au niveau des TGV. En effet, il n'existe pas d'hyperlien capable de prendre un nombre quelconque de contraintes. L'hyperlien conditionnel se rapproche de ces pré-requis. Pour cela, il faut prendre en entrée un ensemble de contraintes (typées), auxquelles nous associons un hyperlien de directionnel, le tout est projeté par l'hyperlien. La liste des contraintes typées est associée à l'hyperlien de typage, seule la contrainte par défaut doit se dissocier des autres contraintes.

Les fonctions nécessitent un typage des éléments d'entrée. A cette fin, une contrainte supplémentaire doit être associée aux nœuds d'entrée et de sortie de la fonction.

5.3 Interface Graphique et Résultats

Interface Graphique. L'interface graphique du médiateur *XLive* (voir figure 5.3) propose des techniques d'affichage graphique permettant de traduire les composants le TGV dans sa représentation graphique. Ainsi, chaque requête reconnue par le médiateur peut alors être représentée sous sa forme graphique et ainsi prouvée sa validité et donner une représentation intuitive de la requête soumise au médiateur. Les TGV correspondant aux requêtes des cas d'usage sont représentés en annexe.

Intégration des TGV. Afin de montrer l'intégration des TGV dans *XLive*, les requêtes du cas d'usage XMP (dont les représentations sont dans l'annexe F) ont été testées sur les sources de données *eXist 1.0* et *Xhive 7.4*. Le tableau 5.11 donne les temps de réponses du médiateur et des sources de données. Les tests ont été réalisés sur un AMD Athlon 1.8GHz, avec 1024Mo RAM sous Windows XP SP2, les sources ont été testées sur un Pentium 2.65GHz, avec 512Mo de RAM sous Windows XP SP2 avec une connection à 10Mbps.

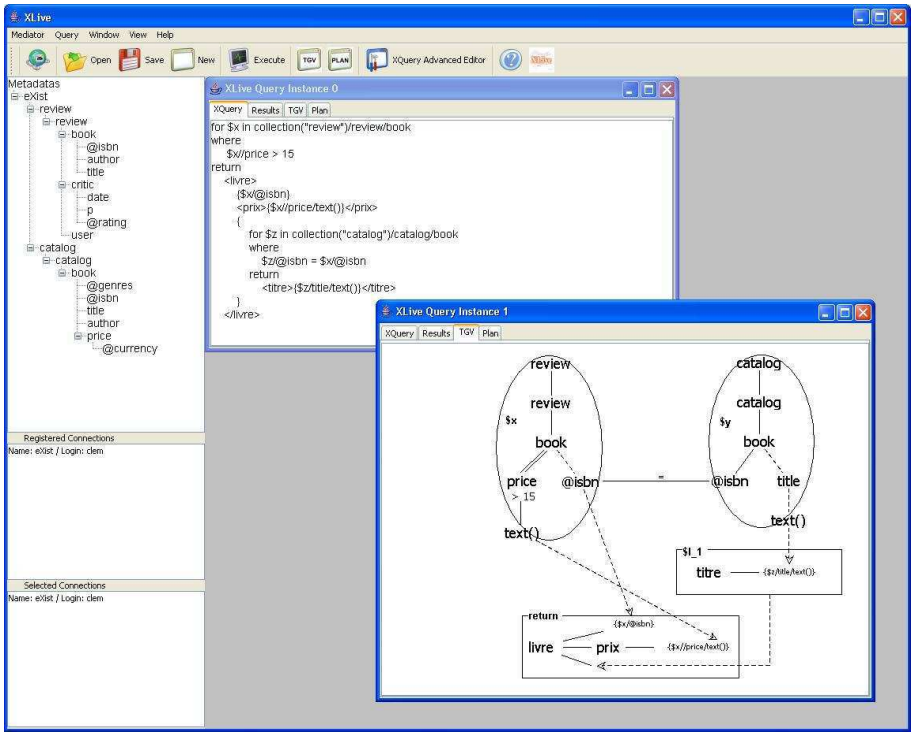


FIG. 5.3 – Capture d'écran de l'interface graphique de XLive

Requête	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
eXist	14,10	13,35	13,30	23,45	14,10	11,70	11,00	11,75	15,60	13,30	13,30	12,45
eXist/XLive	17,20	15,65	16,40	25,00	16,45	17,20	16,40	15,70	19,80	18,75	16,40	16,40
rapport	1,22	1,17	1,23	1,07	1,17	1,47	1,49	1,34	1,27	1,41	1,23	1,32
Xhive	15,65	15,65	17,95		18,60	14,05	15,80	17,95	12,40	10,15	15,70	20,40
Xhive/XLive	21,00	28,20	28,10	17,15	39,00	34,40	32,15	29,60	30,50	21,10	30,55	50,65
rapport	1,34	1,80	1,57		2,10	2,45	2,03	1,65	2,46	2,08	1,95	2,48

TAB. 5.11 – Comparaison des temps d'évaluation des sources eXist et Xhive par rapport au médiateur XLive

Le tableau 5.11 donne le temps d'évaluation des requêtes du cas d'usage XMP (en *ms*). Les tests comparatifs sont fait sur les sources *eXist* et *Xhive*, et chacune d'elle est interrogée par le médiateur *XLive*. Nous pouvons observer un surcoût moyen de **1.28** de la part du médiateur pour la source de données *eXist*, et de **1.99** pour la source de données *Xhive*. Cette différence notoire de rapport entre les deux sources de données provient de la structure des résultats utilisée. En effet, les données retournées par *Xhive* sont encapsulées dans des structures lourdes. Cette structure apporte un surcoût de communication qui détériore les performances du médiateur. Par contre, la source *eXist* donne une simple de chaîne de caractères pour chaque résultat, le médiateur *XLive* est plus efficace pour récupérer des données textuelles.

Le diagramme de la figure 5.4 donne une étude comparative des valeurs du tableau 5.11. Dans la plupart des cas, les temps d'évaluation en *ms* sont proches pour les sources de données *eXist* et *Xhive*. Les requêtes 9 et 10 donnent une évaluation

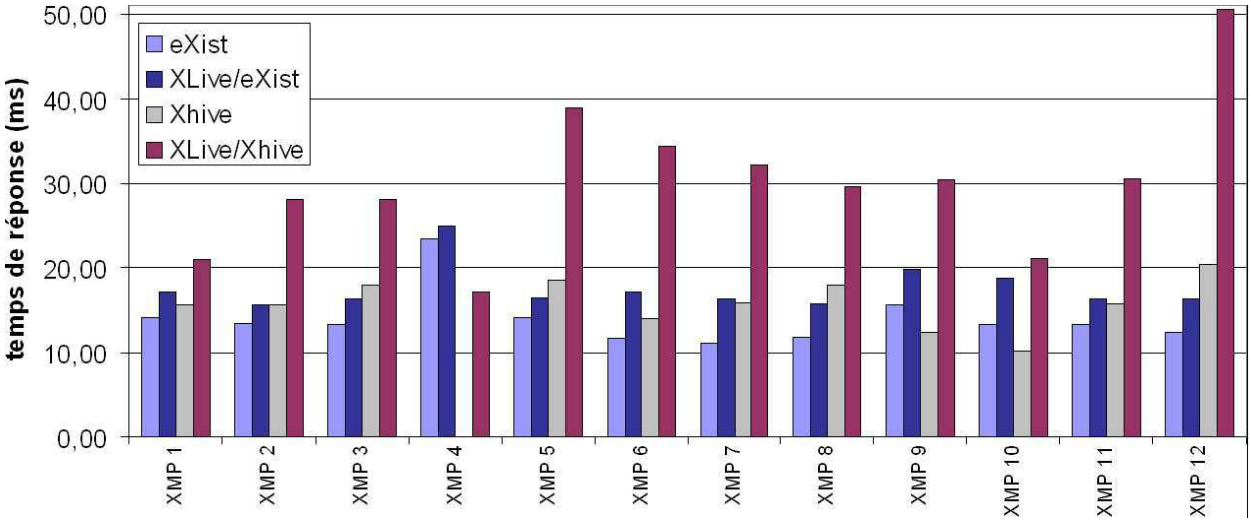


FIG. 5.4 – Performances des sources eXist et Xhive comparées au médiateur XLive, avec le cas d'usage XMP

plus performante pour *Xhive* car celui-ci est plus efficace dans les cas de fonctions d'agrégation de nombreuses valeurs, et pour les recherches de caractères «contains». Nous pouvons constater le fort surcoût de communication pour la source de données *Xhive*, tandis que l'évaluation des requêtes sur *eXist* a un surcoût moindre. Il est flagrant pour les requêtes 5, 6 et 12 pour lesquels les temps de réponse sont deux fois supérieurs à ceux de *Xhive* ; en effet, la taille des données reçues par le médiateur est supérieure aux autres requêtes et demande un surcoût de traitement dans le médiateur. Fait intéressant pour la source Xhive, la requête XMP 4 n'est pas reconnue par cette source, tandis que XLive peut la reconnaître, et l'envoyer à la source Xhive sous une forme qu'elle est capable de reconnaître (quantificateur canonisé).

5.4 Optimisation de Requêtes

L'optimiseur extensible intégré au médiateur permet de manipuler les *TGV*. Les objets Java composant les TGV permettent les opérations de mise à jour (modification, suppression, ajout), utilisées par les règles de transformation. La base d'annotation permettant d'ajouter des informations physiques aux TGV est implémentée de manière à faciliter l'héritage de classes ; pour permettre les différents types d'annotations. Ses annotations permettent d'intégrer les TGV et d'autres annotations pour faciliter la manipulation (voir section 4.2).

Pour générer un arbre algébrique, une annotation pour la *XAlgèbre* est implémentée : *AnnotationXAlgebre*. Elle permet d'associer les XOpérateurs aux éléments du TGV, tout en facilitant la génération de l'arbre d'exécution. La première étape de l'optimiseur est d'annoter le TGV à l'aide des annotations XAlgèbre. Grâce aux correspondances TGV/XAlgèbre, une première annotation est décomposée récursivement sur les éléments du TGV. Chaque annotation contient alors un ou plusieurs

éléments du TGV, ainsi que une ou plusieurs *AnnotationXAlgebre* correspondant aux opérateurs de l'arbre algébrique. Grâce à ces annotations, l'arbre algébrique peut être généré automatiquement.

Le rôle de l'optimiseur est de modifier le TGV et les annotations pour générer l'arbre algébrique dont l'évaluation est optimale. Pour cela, une annotation pour le modèle de coût est ajoutée aux TGV : *AnnotationCostModel*. Celle-ci permet d'annoter une *AnnotationXAlgebre* pour lui associer une formule de coût, définie par les éléments du TGV annoté. Ainsi, nous pouvons obtenir simplement un coût d'évaluation pour chaque arbre algébrique.

Les transformations de l'optimiseur sont définies par les règles de transformation. Ces règles sont appliquées sous condition (composition du TGV et/ou des annotations) et produisent un nouveau TGV, et son annotation de la XAlgèbre correspondante (si nécessaire). Ces règles de transformation sont définies par le langage de règles proposé dans la section 4.3.1. Un parseur développé à l'aide de *JavaCC* permet de générer pour chaque règle une classe *Java* contenant les conditions d'application et les transformations : *Rule*. Chaque classe *Rule* est intégrée à la bibliothèque de règles de l'optimiseur lors de son initialisation.

Pour orienter le choix des règles de transformation, la stratégie de recherche proposée dans la section 4.4.2 doit associé un coefficient d'amélioration à chaque règle. Ainsi, l'optimiseur extensible choisi la règle de transformation applicable dont le coefficient est le plus bas. Ce coefficient est attribué à la classe *Rule* par calibrage sur un ensemble de requêtes définie pour le modèle de coût. Ce coefficient est affecté par le type de chaque règle de transformation, si celle-ci n'est pas définie comme équivalente (lors de sa création), la règle est dite *utilisateur*. Sinon, lorsque la règle manipule des annotations elle est dite *physique*, sinon *logique* sans annotations.

Ainsi, l'optimiseur extensible ordonne les règles d'équivalence selon leur coefficient d'amélioration. A chaque application de règle, il génère l'arbre algébrique correspondant s'il est nécessaire (modification de groupes d'éléments annotés), puis les annotations de coût sont de nouveaux recalculées. Enfin, il recommence le processus au début pour vérifier si la transformation a généré de nouvelles possibilités de transformation. Une fois la fin de la liste atteinte, toutes les transformations possibles ont été appliquées, l'arbre algébrique peut alors être généré à partir des *AnnotationXAlgebre* sur le TGV. L'arbre algébrique obtenu est alors évalué.

5.5 Etudes de performances

Afin de tester les performances de l'optimiseur extensible dans le cadre du médiateur *XLive*, nous évaluons une requête sur le banc d'essai [Dragan et Gardarin 2005]. Ce banc d'essai produit des documents XML de taille variable, dont la répartition est paramétrable. Nous avons choisi des données capables de supporter l'exemple contenu du chapitre 2. Le banc d'essai a été effectué sur des données de différentes

tailles : 100ko, 1.5Mo, 3Mo, 6Mo et 12Mo.

```
for $a in collection ("books")/book
where contains ($a/author, "Robin Hobb")
return
        <book>
                {$a/title}
                {
                        for $b in collection ("catalogs")//book
                        where $a/@isbn = $b/@isbn
                                and some $c in $b/comments
                                        satisfies count($c/comment) > 2
                        return
                                {$b/price}
                                {$b/comments}
                }
        </book>
```

Tab. 5.12 – Requête d'interrogation du banc d'essai

La requête test est présentée dans le tableau 5.12. Cette requête demande les livres écrits par «*Robin Hobb*». Pour ceux-ci, un document est créé contenant le titre du livre, ainsi que les informations correspondantes dans la collection «*catalogs*». Pour cette collection, il doit y avoir plus de deux commentaires à chaque instance. Dans ce cas, le prix de vente du livre est retourné, ainsi que les commentaires. Pour le banc d'essai que nous avons testé, la collection «*books*» est intégrée dans une source *eXist*, tandis que la collection «*catalogs*» est répartie équitablement sur deux sources *Xhive*. Cette répartition permet de montrer les avantages de la médiation avec une jointure des deux collections, ainsi qu'une union des deux sources capables de répondre à une partie de la requête.

Les règles de transformation intégrées à l'optimiseur sont des règles de transformation physique car elles utilisent les informations d'annotation (capacités fonctionnelles des sources pour les fonctions, cardinalité pour l'algorithme de jointure). Ainsi, trois règles d'optimisation sont intégrées, tour à tour, dans l'optimiseur pour montrer son extensibilité :
– déléguer aux sources la fonction *contains* ;
– déléguer aux sources le calcul d'agrégat ;
– changer l'algorithme de jointure (boucle imbriquée vers fonction de hachage).
La figure 5.5 montre les quatre plans d'exécution obtenus durant chaque étape d'optimisation : (a) arbre algébrique initial, (b) délégation de la fonction *contains*, (c) délégation de l'agrégat et (d) modification de l'algorithme de jointure.

Le graphique de la figure 5.6 montre les quatre courbes de performances du médiateur, avec en abscisses la taille des données et en ordonnées les temps de réponse. Chaque courbe est fonction des règles d'optimisation appliquées :
– La première courbe (cercles vides) donne l'évaluation du plan d'exécution sans optimisation ; la quantité de données récupérées influes sur les performances du médiateur ; en effet, les données doivent être filtrées par la fonction «contains» et

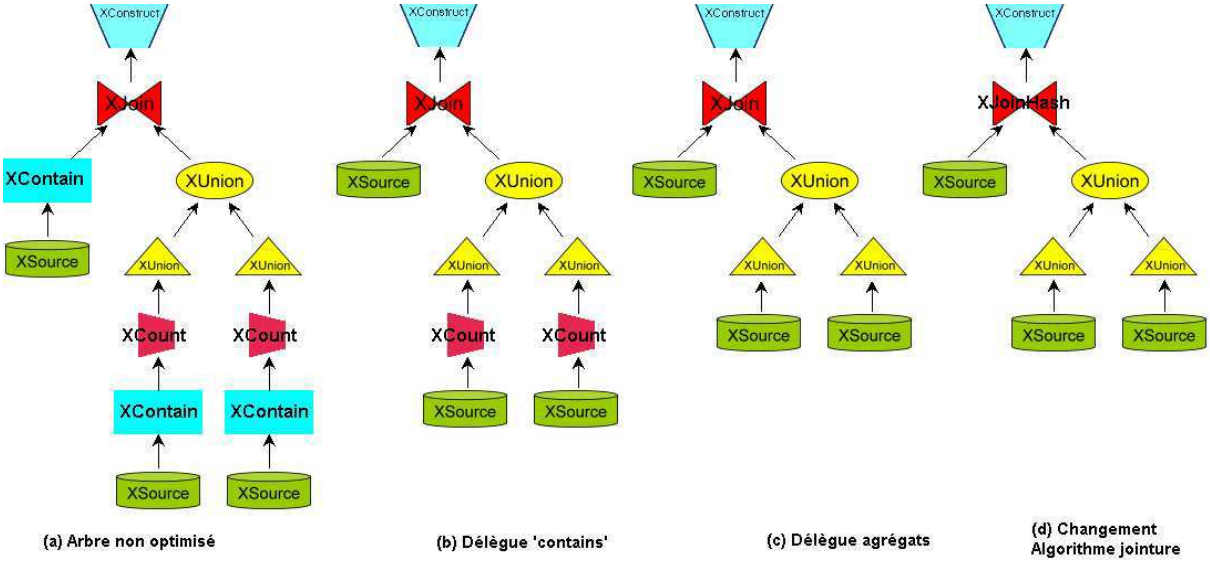


FIG. 5.5 – Étapes d'optimisation de l'arbre algébrique

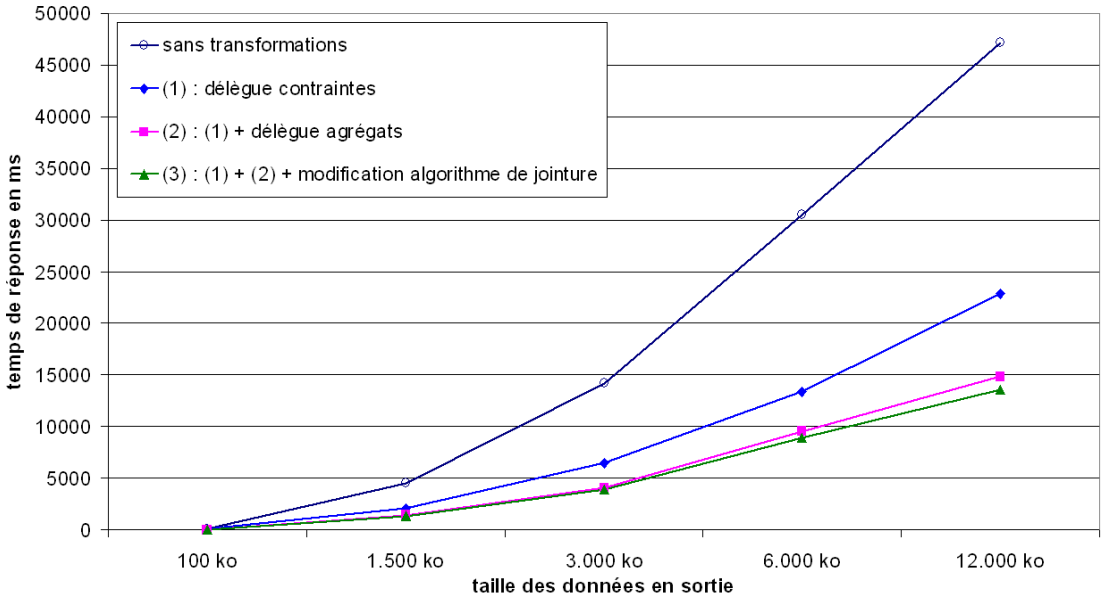


FIG. 5.6 – Graphique de performances de l'optimiseur

par un agrégat pour enfin être jointes ;

- La deuxième courbe (losanges) donne l'application de la première règle d'optimisation (délègue la fonction «contains») ; l'amélioration des performances est due au nombre de données filtrées directement par les sources distantes ;
- La troisième courbe (carrés) donne l'application de la première et de la deuxième règle d'optimisation (délègue «contains» et *agrégats*) ; le nombre de données reçues

par le médiateur améliore efficacement les performances de l'évaluation ;
– La quatrième courbe (triangles) donne l'application des trois règles d'optimisation (en plus le changement de l'algorithme de jointure) ; la quantité de données reçues par le médiateur est identique, mais l'algorithme de jointure améliore légèrement le temps de traitement des données.

Le graphique de la figure 5.7 donne l'évolution effective du coefficient d'amélioration d'une règle d'optimisation. Ce coefficient est obtenu à partir des temps de réponses du graphique 5.6, par le rapport entre le temps avec optimisation sur le temps sans l'optimisation (i.e., temps règle 2 / temps règle 1). Nous pouvons observer que chacune garde en moyenne son coefficient théorique (à gauche sur le graphique), appliqué lors de l'étape d'optimisation. Nous pouvons ainsi constater l'efficacité de l'optimiseur extensible qui permet d'améliorer les performances du système de médiation *XLive* a chaque règle intégrée dans l'optimiseur.

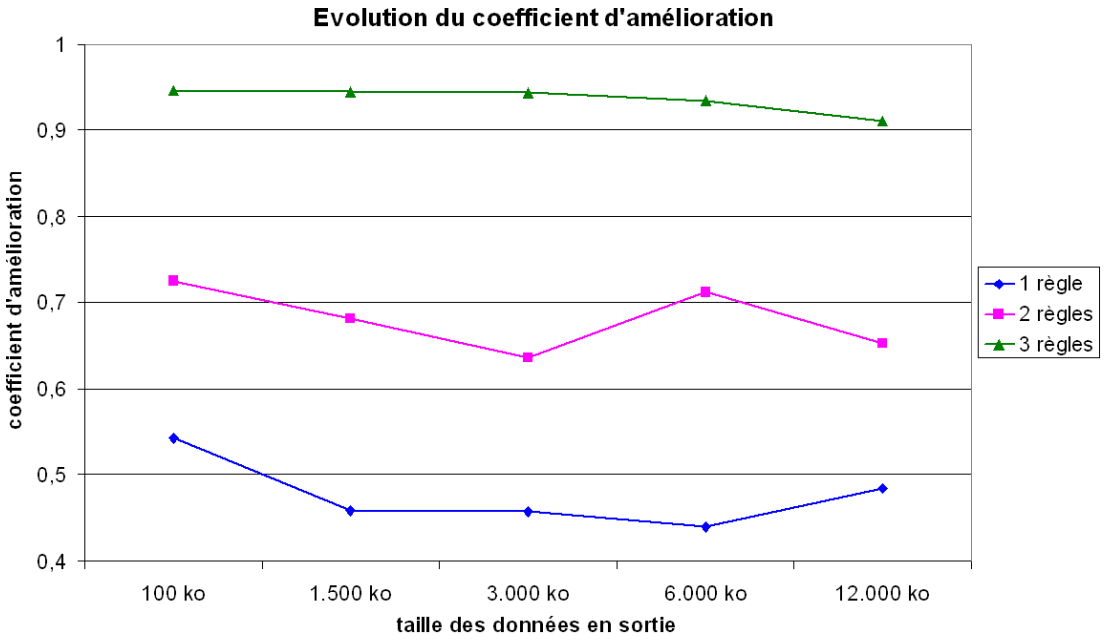


FIG. 5.7 – Graphique d'évolution des coefficients d'amélioration des règles

5.6 Prototypes et Intégration des projets de recherche

Le système de médiation **XLive** permet de fédérer un certains nombres de sources hétérogènes. En effet, les adaptateurs présents dans le médiateur permettent la communication avec différentes sources de données. Les adaptateurs que nous avons implémentés sont les suivants :
– SGBD natif XML : *Xyleme, eXist, Xhive* ;
– SGBD relationnel : *Oracle, MySQL, Microsoft Access* ;
– Fichiers XML ;
– Services Web : *Google API, Amazon.*

La validité du projet *XLive* a été prouvée par son intégration dans quatre projets. Chaque projet a permis le développement de prototype d'application. XLive a ainsi permis d'échanger de nombreuses informations entre sources hétérogènes. Nous résumons ci-dessous ces projets :

- **Projet IST WebSI [WebSI 2004] :** Le projet WebSI (Data-Centric Web Service Integrator) permet de concevoir et de déployer des applications Web gourmandes en données et en contenus. Il consiste en une série d'outils, dont une API s'appuyant sur des standards, qui conçoit une application connectées à des sources d'information hétérogènes. XLive y joue un rôle important de médiation d'information entre l'API et les sources de données *Xyleme* et relationnelles. Les applications sont centrées autour du tourisme en Espagne (*Huelva*) et en Autriche (*Tiscover*).

- **Projet IST Satine [Satine 2004] :** Le projet SATINE a réalisé une plate-forme sécurisée d'échange pour exploiter les services web pour les agences de voyage. Conçu au dessus d'un réseau pair-à-pair (P2P), SATINE permet d'intégrer les services Web grâce à leur description sémantique. Ses applications sont dans le domaine des agences de voyage et de l'ontologie OTA (Open Travel Agency). Xlive est une partie intégrante de la plate-forme puisqu'il permet de fédérer les services web par l'intermédiaire de l'adaptateur Web Service. XLive a permis le développement d'adaptateurs pour le réseau P2P.

- **Projet ACI MD SEMWEB [SemWeb 2004] :** Le projet SemWeb permet le développement d'un web sémantique. L'interrogation de grandes collections de données semi-structurées à l'aide du langage XQuery permet de définir des méthodes, des algorithmes et des architectures pour l'intégration de données hétérogènes et distribuées. Ce projet est basé sur l'utilisation d'architectures pair-à-pair à base de médiation. XLive se trouve au centre de l'architecture, permettant de fédérer les nombreuses sources hétérogènes présents sur le web (résumés de données, multistructuration de documents XML, services web, réplication de données).

- **Projet ANR PADAWAN [Padawan 2005] :** Le projet PADAWAN (Proxy for All Devices Accessing the World And Neighbourhood) a pour but de fournir à un utilisateur mobile ou non, l'accès à un Proxy Universel. Ce proxy doit permettre l'intégration des données provenant de sources très hétérogènes : SGBD relationnel, XML, des sites web, des capteurs (localisation GPS, température, pression, caméra, etc.), des applications. XLive y jour un rôle prépondérant de médiation d'informations hétérogènes, fournissant les résultats homogènes au proxy universel. Il a permit l'intégration d'un adaptateur pour des informations provenant de capteurs, demandant une traduction particulière des requêtes ainsi qu'une harmonisation des résultats fournis par les capteurs.

5.7 Conclusion

Le médiateur *XLive* permet d'évaluer des requêtes XQuery à l'aide de la *XAlgèbre*. Les arbres d'*XOpérateurs* sont produits par transformation de TGV annotés. Grâce à l'évaluation de requêtes, les cas d'usage du W3C ont pu être validés, de manière qualitative et quantitative. De plus, à l'aide d'un banc d'essai [Dragan et Gardarin 2005], nous avons pu testé les performances du médiateur avec des règles de transformation qui ont été ajoutées dans l'optimiseur.

Un des points forts du médiateur *XLive* est de pouvoir reconnaître une majeure partie des requêtes *XQuery* qui peuvent lui être soumises. En effet, grâce au modèle *TGV*, le parseur de requêtes peut reconnaître toutes les requêtes *XQuery* non typées. La validation par huit des neufs cas d'usage proposés par le W3C permet de donner une idée des capacités du médiateur. De plus, l'extension du modèle pour intégrer les typages reste possible grâce au typage des opérateurs et de leurs paramètres. Les opérateurs de la XAlgèbre ne tiennent pas compte, pour le moment, du typage des données. L'intégration des types demanderait une restructuration importante de l'algèbre.

Enfin, l'optimiseur extensible présent dans le médiateur permet à celui-ci de s'adapter grâce à la configuration de l'optimiseur en fonction des besoins des requêtes et des sources de données. Ainsi, l'utilisation des règles de transformation peut être orientée pour une forte délégation des opérations aux sources de données et pour des changements d'algorithme des XOpérateurs. L'optimiseur extensible permet d'adapter le médiateur à de nombreux projets dont les pré-requis sont variés.

Chapitre 6

Conclusion

Dans cette thèse, nous avons proposé une approche complète de traitement de requêtes dans un contexte de médiation de données semi-structurées. Le langage de requête pris en compte est XQuery, langage très complet mais complexe. Nous avons défini une architecture extensible pour interroger des sources variées en XML/XQuery. Nous avons intégré nos résultats dans le système de médiation de données *XLive* [Dang-Ngoc *et al.* 2005] que nous avons en partie réalisé puis étendu. L'étude des caractéristiques de la plupart des architectures de médiation opérant sur des sources de données variées que nous avons réalisée montre que très peu de systèmes de médiation sont aujourd'hui basés sur XQuery. Les méthodes d'interrogation de ces systèmes doivent tenir compte du langage propre à chaque source, donc traduire au moins en partie XQuery dans les langages des sources. Ceci est sans doute une des difficultés de la médiation «tout XML». Le système de médiation doit donc être capable de communiquer avec chacune des sources. Les outils capables de traduire les requêtes sont appelés des adaptateurs. Notre architecture est capable de supporter une bonne variété d'adaptateurs plus ou moins complets au regard de XQuery. Ceux-ci permettent le dialogue entre la source et le médiateur, ils peuvent aussi fournir des informations supplémentaires sur la source de données pour alimenter l'optimiseur extensible.

Afin de traiter les requêtes soumises au médiateur, un modèle de représentation de requêtes est nécessaire. En effet, la localisation des sources, la séparation des traitements, l'optimisation et l'évaluation nécessitent un prétraitement s'appuyant sur un modèle de représentation des requêtes. Celui-ci doit intégrer les caractéristiques du langage d'interrogation, les problèmes liés à l'architecture pour faciliter les transformations et aussi permettre une étape d'optimisation. Dans le contexte de médiation, le modèle doit pouvoir séparer les sources de données, les opérations de traitement pour identifier où doit s'exécuter chacune d'elles, définir les opérations que le médiateur doit exécuter, ajouter des opérations pour intégrer les informations provenant des sources. Tout ceci a pour objectif de préparer une évaluation efficace des requêtes.

Dans un souci d'amélioration des performances lors de l'évaluation des requêtes, une étape d'optimisation des requêtes a été proposée dans cette thèse. De plus, afin d'accroître le potentiel de l'optimisation dans le contexte de la médiation, nous nous sommes concentrés sur les architectures d'**optimiseur extensible**. Nous avons donc étudié les différentes techniques permettant de définir et réaliser l'extensibilité d'un optimiseur. Nous avons proposé un langage de règles simple et visuel pour décrire les transformations de requêtes possibles. Au total, le processus d'optimisation est décomposé en différentes étapes (modélisation de la requête, règles de transformation, stratégie de recherche et modèle de coût) ; il permet de manipuler la représentation originale des requêtes pour obtenir un plan d'exécution proche de l'optimal

Enfin, nous avons validé nos travaux à l'aide d'une évaluation des cas d'usage du [W3C 2006b], ce qui nous a permis de montrer l'expressivité des TGV. Puis, des tests ont été effectués sur le médiateur *XLive* à l'aide des bancs d'essai de [Dragan et Gardarin 2005], permettant d'étudier les performances de l'optimiseur extensible. Pour finir, la validité du médiateur *XLive* a été prouvé par son intégration dans des projets fonctionnels : [WebSI 2004], [Satine 2004], [SemWeb 2004] et [Padawan 2005].

6.1 Comparaison avec les approches existantes

Dans le chapitre d'état de l'art (chapitre 2), nous avons présenté les divers systèmes de médiation. Nous avons étudié les techniques de représentation de requêtes proposées pour le langage XQuery. Puis, nous avons revu différents optimiseurs extensibles existants. Nous avons alors caractérisé chacune de ces approches pour bien comprendre ces différentes techniques. Maintenant, nous positionnons notre travail par rapport à chacune de ces approches, tout en gardant à l'esprit qu'aucune d'elles n'abordent les problématiques liées à notre contexte global de médiation «tout XML».

6.1.1 Les systèmes de médiation

Parmi les systèmes de médiation présentés, seul le système *LiquidData* [BEA 2002] aborde une architecture «tout-XML» en utilisant un modèle de données basé sur XML, et une reconnaissance des requêtes XQuery. A notre connaissance, le parseur de *LiquidData* propose de reconnaître la plupart des requêtes XQuery, mais nous pouvons indiquer qu'il ne peut reconnaître ni les séquences, ni les opérations conditionnelles, ni les typages complexes (*typeswitch*). Par contre, *LiquidData* est capable de reconnaître les typages liés à la structure des méta-données, relevant des opérations '*as*' et '*instance of*'. La modélisation de leurs requêtes ne consiste qu'à l'affichage du plan physique résultant, qui peut être optimisé manuellement. L'optimiseur intégré à ce système de médiation est très simple. En effet, il ne repose que sur une optimisation statique du plan physique. Un choix des algorithmes des opérateurs

de l'algèbre XML utilisée et un ordonnancement des sources permet d'améliorer les performances de l'évaluateur.

Ainsi, notre proposition est un système parallèle aux propositions de *LiquidData*, la grammaire reconnue est plus large, la modélisation aborde une approche plus logique et intuitive. De plus, l'optimiseur que nous proposons aborde des problèmes plus en profondeur concernant le contexte de médiation. En addition, il est paramétrable et peut être étendu grâce à l'extensibilité de ses règles de transformation.

6.1.2 Les représentations de requêtes

L'étude des techniques de représentation de requêtes sur les motifs d'arbre analyse la proposition des *Tree Pattern Queries* [Jagadish *et al.* 2001][Amer-Yahia *et al.* 2001][Chen 2004][Manolescu *et al.* 2004](section 2.2.2). Ce modèle développé en parallèle au TGV propose une représentation intéressante des XQuery. Malheureusement, cette proposition et les similaires [Arion *et al.* 2005a][Arion *et al.* 2006][Benzaken *et al.* 2004] abordent une approche purement physique, très proche de l'implémentation dans un SGBD XML. En effet, les représentations ne s'intéressent qu'au filtrage des documents, et très peu aux autres caractéristiques du langage. Celles-ci ne sont utilisées que dans des opérateurs de l'algèbre et ne sont donc pas représentées. Ainsi, la représentation d'une requête XQuery se cantonne au filtrage de documents avec prédicats.

Notre modèle généralise les techniques existantes pour proposer une approche logique, en abordant chacune des caractéristiques du langage et en les intégrant dans le modèle. Nous obtenons ainsi un modèle permettant de reconnaître la plupart du langage XQuery et d'offrir une représentation logique et intuitive. Il généralise les principes apportés par les *GTP* tel que les motifs d'arbre et la canonisation de requêtes.

6.1.3 Les optimiseurs extensibles

Les différents optimiseurs extensibles que nous avons étudiés ne s'intègrent, à notre connaissance, qu'à des SGBD relationnels ou orientés-objets. La plupart des techniques sont intéressantes et ont inspiré notre travail. En effet, le langage de définition de règles de transformation défini dans la section 4.3.1.1 ressemble à celui-ci proposé dans le projet *ESPRIT EDS* [Finance et Gardarin 1991]. Notre stratégie de recherche repose sur les techniques utilisées dans les architectures *EXODUS/Volcano* [Graefe et DeWitt 1987][Graefe 1987][Carey *et al.* 1990][Graefe et McKenna 1991][Graefe et McKenna 1993] (coefficient d'amélioration) et *OPT++* [Kabra et DeWitt 1999] (optimisation du plan logique et du plan physique).

Notre optimiseur extensible est donc un mélange intéressant de techniques permettant de faciliter l'extensibilité de l'espace de recherche orienté vers un contexte

de médiation. Il présente l'originalité d'être le premier optimiseur extensible pour XML/XQuery, à notre connaissance.

6.1.4 Synthèse

Le système de médiation *XLive* intègre ainsi un ensemble de techniques pour donner une solution de médiation originale et complète. Il permet la médiation de requêtes XQuery en utilisant une architecture «tout-XML». Il propose une représentation de requête logique et intuitive permettant d'intégrer les solutions aux problèmes de médiation et d'optimisation. L'optimiseur extensible proposé dans ce système est basé sur un ensemble de règles qui peut être enrichi, permettant au médiateur de s'adapter au mieux à son domaine et ainsi de configurer un optimiseur optimal en fonction des sources de données reliées au médiateur.

Le modèle de représentation de requêtes n'est ni spécifique à la médiation, ni spécifique à XQuery. En effet, les TGV permettent de représenter une requête et proposent une base d'évaluation ; ils peuvent donc être intégrés à n'importe quel SGBD natif XML. De plus, le langage XQuery n'est pas le seul langage représentable car le modèle proposé introduit une structure de données, ainsi que des liens entre structures. Tout autre langage sur des données XML, orientées-objets ou relationnelles peut être représenté dans ce modèle, à partir du moment où il ne demande pas de typage d'éléments.

Enfin, la base d'annotation des *TGV* permet au modèle de s'adapter aux demandes extérieures pouvant s'appuyer sur les requêtes XQuery. Puisqu'une annotation peut représenter n'importe quelle contrainte en spécialisant des ensembles caractéristiques, il est alors possible de modéliser toute sorte d'information sur notre modèle.

6.2 Contributions

Dans cette thèse, nous avons donc proposé une approche de traitement de requêtes dans un contexte de médiation de données semi-structurées. Le langage de requêtes XQuery a orienté notre réflexion pour proposer un modèle de représentation adapté pour ce contexte et faciliter les manipulations dans le cadre de l'optimisation. L'ensemble des contributions est présenté dans le journal [Travers *et al.* 2006]. Elles détails les différentes étapes suivantes :

1. Canonisation ;
2. Tree Graph Views ;
3. Optimisation extensible ;
4. Médiation.

6.2.1 Canonisation

Avant de traduire une requête XQuery dans notre modèle, une étape de préparation transforme la requête. La section 3.2 présente cette étape de transformations sous la forme d'une *canonisation* des requêtes. Un ensemble de règles permet de préparer certaines caractéristiques du langage pour les transformer en une expression qui peut être représentée dans notre modèle. Ces résultats étendent ceux de [Chen 2004].

6.2.2 Tree Graph Views

Notre modèle de représentation de requêtes *TGV*, présenté dans le chapitre 3, est composé d'éléments correspondant aux caractéristiques du langage XQuery. La section 3.3 donne la définition des éléments composant notre modèle. Il est composé de nœuds correspondant aux éléments d'un document XML, des liens entre nœuds représentant les axes de parcours dans le document XML, des contraintes à appliquer aux nœuds ou leur contenu pour filtrer le document. Un ensemble de nœuds et de liens de nœuds forment un motif d'arbre appelé *Tree Pattern* dérivés de [Amer-Yahia *et al.* 2001]. Des hyperliens relient les éléments entre eux pour former des expressions caractéristiques du langage XQuery.

Notre modèle a été formalisé à l'aide des *Types Abstraits de Données* dans la section 3.4. Chaque élément des TGV montre l'ensemble des opérations qui le compose et les interactions nécessaires pour relier les éléments entre eux.

Enfin, la section 3.5 propose un algorithme de traduction des requêtes XQuery, permettant de faire correspondre chaque caractéristique d'une requête XQuery canonique avec le modèle de représentation de requêtes *TGV*.

6.2.3 Optimisation extensible

Dans le but d'améliorer les performances d'évaluation des requêtes dans notre médiateur de données, un optimiseur a été proposé dans le chapitre 4. Avant d'aborder l'étape d'amélioration du processus, une base d'évaluation a été proposée. Cette *algèbre abstraite* (section 4.1) donne la manière d'évaluer un TGV sur un ensemble de documents XML. Basée sur des opérateurs logiques d'évaluation, chaque propriété du langage XQuery est traduite à l'aide de ces opérateurs.

6.2.3.1 Annotation

Pour différencier les optimisations logiques et physiques, une *base d'annotation* (section 4.2) des TGV permet d'étendre le TGV logique (sans annotations) en TGV physique (avec annotations). Ces annotations permettent d'ajouter des informations

supplémentaires aux TGV, tels que, des choix d'algorithmes pour des opérations de l'algèbre abstraite, des regroupements d'ensembles d'élément ou des formules de coût. Toutes ces annotations enrichissent le modèle pour lui donner un aspect physique qui pourra être optimisé.

6.2.3.2 Langage de règles

Afin de permettre à l'optimiseur d'être paramétrable et de particulariser l'optimisation en fonction de l'environnement, un *langage* de définition *de règles de transformation* est proposé. Il permet d'ajouter de nouvelles règles dans l'optimiseur. Ce langage (section 4.3) de notre modèle est défini sous forme de Types Abstraits. Une transformation sur un TGV ne peut être appliquée que si elle préserve l'équivalence, c'est-à-dire que le résultat de l'évaluation doit être identique avant et après la transformation. Une catégorisation des règles permet de différencier l'optimisation logique (sans annotations), de l'optimisation physique (avec annotations). De plus, pour permettre de personnaliser l'optimiseur, des règles utilisateurs permettent à l'administrateur du médiateur de le paramétrer spécifiquement.

6.2.4 XLive

Pour permettre une évaluation physique des données dans un médiateur, le système de médiation *XLive* a été implémenté (chapitre 5). Il intègre le modèle *TGV*, une algèbre relationnelle étendue à XML (*XAlgèbre* [Dang-Ngoc 2003]), l'*optimiseur extensible* et des adaptateurs pour des sources variées. Le cycle de traitement des requêtes XQuery proposé est intégralement implémenté dans le médiateur : *modélisation*, *optimisation* et *évaluation*.

Les cas d'usage du W3C ont été la base de test pour prouver la validité de notre modèle. Sur les neufs catégories de cas d'utilisation proposées, huit sont reconnues. Le typage n'étant pas encore intégré au modèle, le cas d'utilisation *STRONG* ne peut être reconnu, une solution est toutefois proposée pour étendre le modèle au pré-requis du langage.

Le système de médiation *XLive* a été au cœur de projets européens IST ([WebSI 2004] et [Satine 2004]), ainsi qu'un projet ACI MD ([SemWeb 2004]). Ce système «tout-XML» facilite la fédération de données dans un modèle d'échange standard. Il s'intègre facilement dans de nombreux cas.

6.3 Perspectives de recherche

A court terme, nous pensons étendre le travail réalisé dans cette thèse :

- Nous envisageons d'intégrer le support du typage du langage XQuery. Notre modèle de représentation de requêtes *TGV* sera alors complet, et pourra être validé par tous les cas d'usage. Voici le détail des différentes opérations de typage présente dans les spécifications du W3C :
  - l'opération «*as*» est à la fois un contrainte vérifiant le typage de l'élément, mais aussi une définition de type pour l'élément qui est vérifiée tout au long d'une requête XQuery. Il faut donc étendre la définition du nœd des TGV pour y intégrer la définition de typage, représentée sous le nœud comme une contrainte ;
  - l'opération «*castable*» vérifie si un élément peut être typé. Elle peut être traitée comme une contrainte ;
  - les opérations de vérification du typage «*Kind-test*» (ex. : «*element(*, xs :car)*») sont des contraintes applicables sur les nœuds des motifs d'arbre. Certaines de ces expressions peuvent être canonisées pour faire correspondre la contrainte à une simple vérification de nom d'élément (ex. : *element(car)* devient *car*). D'autres pourront être canonisées pour obtenir un nom d'élément associée à la contrainte «*as*» (ex. : *element(peugeot, xs :car)* devient *peugeot as car*).
  - l'opération «*cast as*» est une transformation de type d'un élément. Il peut correspondre à un hyperlien directionnel donnant le typage de l'élément ;
  - l'opération «*typeswitch*» pose un problème de typologie au niveau des TGV. En effet, cet opération donne un résultat en fonction du type de l'élement en entrée, plusieurs contraintes doivent alors être traitées. Nous pouvons la considérer comme une transformation, équivalent à un hyperlien dans notre modèle. Toutefois, il n'existe pas d'hyperlien capable de prendre un nombre quelconque de contraintes en entrée. L'hyperlien conditionnel se rapproche de ces pré-requis. Pour cela, il faut prendre en entrée un nœud et un ensemble de contraintes de typage, auxquelles nous associons un hyperlien de directionnel, le tout est projeté par l'hyperlien. La liste des contraintes typées est associée à l'hyperlien de typage, seule la contrainte par défaut doit se dissocier des autres contraintes ;
  - d'après les spécifications du W3C, l'opération «*instanceof*» peut être normalisée en opération «*typeswitch*» avec une seule contrainte de typage retournant vrai si la contrainte est vérifiée, et faux sinon ;
  - d'après les spécifications du W3C, l'opération «*treat as*» peut être normalisée en opération «*typeswitch*» avec une seule contrainte de typage («*as*») sur l'élément, si la contrainte est vérifiée il retourne l'élément, sinon, une erreur est générée.
  - les *fonctions* nécessitent un typage des éléments en entrée et en sortie. A cette fin, des contraintes supplémentaires doivent simplement être associées aux nœuds d'entrée et de sortie de la fonction.

  Un exemple de représentation de requête typée est proposée en annexe G.1.

- Le W3C propose d'ajouter les requêtes de mises à jour au standard de XQuery [W3C 2006c]. Nous proposons de créer une modélisation permettant de représenter ces mises à jour. Ainsi, nous serons capable de d'envoyer des requêtes de mises à jour sur les sources. Pour cela, il nous faut intégrer les opérations «IN-

SERT INTO/AFTER/BEFORE», «DELETE», «REPLACE» et «RENAME». Ces opérations manipulent des expressions. Pour définir l'opération de mise à jour, il nous faut concevoir de nouveaux types d'hyperliens (non directionnels car non injective). Nous proposons de définir des flèches de nature différentes dont la cible est l'élément à mettre à jour. Un exemple de représentation de requête de mise à jour est proposée en annexe G.2.

– Nous pouvons ensuite affiner la stratégie de recherche de l'optimiseur extensible. La stratégie utilisée dans le système EPOQ [Bjørnestad 1994] propose de créer des régions d'optimisation, composées d'ensemble de règles, ainsi, nous pourrions orienter l'application des règles en créant des méta-règles d'optimisation. L'étape d'optimisation en serait grandement améliorée en imposant quelques séquences optimisées. Le but serait de composer les règles déjà ajoutées à l'optimiseur en les combinant dans de nouvelles règles d'optimisation.

– Le modèle de représentation *TGV* permet de représenter graphiquement une requête XQuery. Afin de faciliter l'interrogation des documents XML, une interface de création de TGV pourrait aider l'utilisateur à créer intuitivement sa requête au-dessus du médiateur. De plus, l'utilisation des métadonnées des sources faciliterait la création des motifs d'arbre en proposant des chemins valides.

– En partant du même principe, nous pouvons proposer une interface de création des règles de transformation en utilisant la représentation sous forme de motifs de règles. Ainsi, la génération des règles de transformation serait moins complexe puisque l'interface générerait le code adéquate. La configuration de l'optimiseur en serait d'autant plus faciliter.

A plus long terme, nous avons déterminé trois axes majeurs de recherche :

– Il faudrait tout d'abord étendre notre étude aux gestions de vues matérialisées en utilisant le modèle TGV. L'interrogation de vues ne reviendrait qu'à appliquer un simple TGV, tandis que les vues non matérialisées nécessiteraient la fusion de TGV, déjà en partie réalisée. Les mise à jour seraient facilitées par l'application des TGV sur les éléments nécessaires à cette mise à jour [Abiteboul *et al.* 1998][Abiteboul 1999][El-Sayed *et al.* 2002][Dang-Ngoc *et al.* 2004b].

– Un autre axe de recherche possible consisterait à intégrer les TGV dans les réseaux pair-à-pair pour permettre une évaluation distribuée et un routage intelligent des TGV soumis au réseau. Une décomposition des TGV permettrait de diffuser des sous-requêtes qui seraient recomposées par la suite. [Abiteboul *et al.* 2006] propose une manipulation de motifs d'arbre tout en préservant les identifiants des pairs produit par une DHT.

– Enfin, un dernier axe de recherche serait de proposer des annotations spécifiques pour les TGV pour ainsi intégrer de nombreuses technologies basées sur XML et XQuery (données multi-structurées, XPath flous, indexation...). Pour les requêtes floues [Damiani et Tanca 2000][Damiani *et al.* 2000][Calmès *et al.* 2003].

Bibliographie

[Abiteboul *et al.* 1997] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, et Janet L. Wiener. The Lorel Query Language for Semi-Structured Data. *International Journal on Digital Libraries*, 1(1) :68–88, 1997.

[Abiteboul *et al.* 1998] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, et Janet L. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 38–49, 24–27 1998.

[Abiteboul *et al.* 2006] Serge Abiteboul, Ioana Manolescu, et Nicoleta Preda. Sharing Content in Structured P2P Networks. In *BDA*, 2006.

[Abiteboul 1999] Serge Abiteboul. On Views and XML. In *PODS*, pages 1–9, 1999.

[Adali *et al.* 1996] S. Adali, K. Candan, et Y. Papakonstantinou. Query Caching and Optimization in Distributed Mediator Systems. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 137–148, Montreal, Canada, 1996.

[Adjiman *et al.* 2004] P. Adjiman, Philippe Chatalic, François Goasdoué, Marie-Christine Rousset, et Laurent Simon. Distributed reasoning in a peer-to-peer setting. pages 945–946, 2004.

[Aguilera *et al.* 2000] Vincent Aguilera, Sophie Cluet, Pierangelo Veltri, Dan Vodislav, et Fanny Wattez. Querying XML documents in Xyleme. *In Workshop ACM-SIGIR*, 2000.

[Ali et Moerkotte 2004] Robin Ali et Guido Moerkotte. *Query Rewriting with Coko-Kola*. Technical report, University of Mannheim, April 2004.

[Amann *et al.* 2002] Bernd Amann, Catriel Beeri, Irini Fundulaki, et Michel Scholl. Querying xml sources using an ontology-based mediator. In *CoopIS/DOA/ODBASE*, pages 429–448, 2002.

[Amer-Yahia *et al.* 2001] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, et Divesh Srivastava. Minimization of Tree Pattern Queries. In *SIGMOD Conference*, pages 497–508, 2001.

[Andrei et Valduriez 2001] Mihnea Andrei et Patrick Valduriez. User-Optimizer Communication using Abstract Plans in Sybase ASE. pages 29–38, 2001.

[Arion *et al.* 2005a] Andrei Arion, Véronique Benzaken, et Ioana Manolescu. XML Access Modules : Towards Physical Data Independence in XML Databases. In *XIME-P*, 2005.

[Arion *et al.* 2005b] Andrei Arion, Véronique Benzaken, Ioana Manolescu, et Ravi Vijay. ULoad : Choosing the Right Storage for Your XML Application. In *VLDB*, pages 1330–1333, 2005.

[Arion *et al.* 2006] Andrei Arion, Véronique Benzaken, Ioana Manolescu, Yannis Papakonstantinou, et Ravi Vijay. Algebra-Based Identification of Tree Patterns in XQuery. In *FQAS*, pages 13–25, 2006.

[BEA 2002] BEA. BEA Liquid Data for WebLogic - Product Overview - http ://edocs.bea.com, October 2002.

[Benzaken *et al.* 2003] Véronique Benzaken, Giuseppe Castagna, et Alain Frisch. Cduce : An xml-centric general-purpose language. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming. ACM Press*, 2003.

[Benzaken *et al.* 2004] Véronique Benzaken, Giuseppe Castagna, et Cédric Miachon. CQL : a pattern-based query language for XML. In *BDA*, 2004.

[Benzaken *et al.* 2005] Véronique Benzaken, Giuseppe Castagna, et Cédric Miachon. A Full Pattern-Based Paradigm for XML Query Processing. *PADL*, pages 235–252, 2005.

[Biron et Malhotra 2000] Paul V. Biron et Ashock Malhotra. XML Schema Part 2 : Datatypes, Octobre 2000. http ://www.w3.org/TR/2000/CR-xmlschema-2-20001024/.

[Bjørnestad 1994] Solveig Bjørnestad. *The framework for EPOQ - an Extensible Object-Oriented Query Optimizer.* Technical report, Departement of Information Science, University of Bergen, March 1994.

[Bray *et al.* 1997] Tim Bray, Jean Paoli, et C. M. Sperberg-McQueen. Extensible Markup Language (XML). *World Wide Web Journal*, 2(4) :27–66, 1997.

[Brill *et al.* 1984] David Brill, Marjorie Templeton, et Clement T. Yu. Distributed query processing strategies in mermaid, a frontend to data management systems. In *ICDE*, pages 211–218, 1984.

[Buneman 1997] Peter Buneman. Semistructured data. pages 117–121, 1997.

[Calmès *et al.* 2003] Martine De Calmès, Henri Prade, et Florence Sedes. Requêtes flexibles et données semi-structurées. In *LFA*, pages 23–30, 2003.

[Caravel 1998] Caravel. LeSelect, 1998. http ://www-caravel.inria.fr/Faction_Le_Select.html.

[Carey *et al.* 1990] Michael J. Carey, David J. DeWitt, Goetz Graefe, D. M. Haight, Joel E. Richardson, Daniel T. Schuh, Eugene J. Shekita, et Scott L. Vandenberg. The EXODUS Extensible DBMS Project : An Overview. In D. Maier and S. Zdonik, editor, *Readings on Object-Oriented Database Sys.* Morgan Kaufmann, San Mateo, CA, 1990.

[Carey *et al.* 2000a] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene J. Shekita, et Subbu N. Subramanian. XPERANTO : Publishing Object-Relational Data as XML. In *WebDB (Informal Proceedings)*, pages 105–110, 2000.

[Carey *et al.* 2000b] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, et Subbu N. Subramanian. XPERANTO : Middleware for Publishing Object-Relational Data as XML Documents. In *VLDB*, pages 646–648, 2000.

[Carey et Haas 1990] Michael J. Carey et Laura Haas. Extensible Database Management Systems. In *SIGMOD Record*, pages 19(4) :54–60, 1990.

[Carey 2004] Michael J. Carey. BEA Liquid Data for WebLogic : XML-Based Enterprise Information Integration. In *ICDE*, pages 800–803, 2004.

[Chamberlin *et al.* 2000] Donald D. Chamberlin, Jonathan Robie, et Daniela Florescu. Quilt : An XML Query Language for Heterogeneous Data Sources, 2000.

[Chawathe *et al.* 1994] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, et Jennifer Widom. The TSIMMIS Project : Integration of Heterogeneous Information Sources. In *IPSJ*, pages 7–18, 1994.

[Chen 2004] Zhimin Chen. From Tree Patterns to Generalized Tree Patterns : On Efficient Evaluation of XQuery, 2004. University of British Columbia, MSc Thesis, http ://www.cs.ubc.ca/nest/dbsl/thesis/zm_thesis.pdf.

[Cherniack et Zdonik 1998] Mitch Cherniack et Stanley B. Zdonik. Changing the Rules : Transformations for Rule-Based Optimizers. In *SIGMOD Conference*, pages 61–72, 1998.

[Christophides *et al.* 1994] Vassilis Christophides, Serge Abiteboul, Sophie Cluet, et Michel Scholl. From Structured Documents to Novel Query Facilities. In *SIGMOD Conference*, pages 313–324, 1994.

[Cluet *et al.* 1998] Sophie Cluet, Claude Delobel, Jérôme Siméon, et Katarzyna Smaga. Your Mediators Need Data Conversion ! In *SIGMOD Conference*, pages 177–188, 1998.

[Cover *et al.* 1991] Robin Cover, Nicholas Duncan, et Dadid Barnard. *Bibliography on SGML (Standard Generalized Markup Language) and Related Issues*. Technical report, Queen's University, Kingston, Ontario, February 1991.

[Damiani *et al.* 2000] Ernesto Damiani, Letizia Tanca, et Francesca Arcelli Fontana. Fuzzy XML queries via context-based choice of aggregation. *Kybernetika*, 36, 2000.

[Damiani et Tanca 2000] Ernesto Damiani et Letizia Tanca. Blind Queries to XML Data. In *DEXA*, pages 345–356, 2000.

[Dang-Ngoc *et al.* 2004a] Tuyet-Tram Dang-Ngoc, Georges Gardarin, et Nicolas Travers. Tree graph views : On efficient evaluation of xquery in an xml mediator. In *BDA*, 2004.

[Dang-Ngoc *et al.* 2004b] Tuyet-Tram Dang-Ngoc, Virginie Sans, et Dominique Laurent. Classifying XML Materialized views for their maintenance on distributed Web sources. *RNTI-E-3*, II :433–444, 2004.

[Dang-Ngoc *et al.* 2005] Tuyêt-Trâm Dang-Ngoc, Clément Jamard, et Nicolas Travers. XLive : An XML Light Integration Virtual Engine. *BDA*, pages 399–404, October 2005.

[Dang-Ngoc 2003] Tuyêt-Trâm Dang-Ngoc. *Fédération de données semi-structurées avec XML*. PhD thesis, Université de Versailles-Saint-Quentin-En-Yvelines, Juin 2003.

[Deutsch *et al.* 1998] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, et Dan Suciu. XML-QL. In *QL*, 1998.

[Deutsch *et al.* 2004] Alin Deutsch, Yannis Papakonstantinou, et Yu Xu. The NEXT Framework for Logical XQuery Optimization. In *VLDB*, pages 168–179, 2004.

[Dragan et Gardarin 2005] Florin Dragan et Georges Gardarin. Benchmarking an XML Mediator. In *ICEIS (1)*, pages 191–196, 2005.

[Du *et al.* 1992] W. Du, R. Krishnamurthy, et M.-C. Shan. Query Optimization in a Heterogeneous DBMS. In *Proc. of the 18th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 277–291, Vancouver, Cananda, 1992.

[El-Sayed *et al.* 2002] Maged El-Sayed, Ling Wang, Luping Ding, et Elke A. Rundensteiner. An algebraic approach for incremental maintenance of materialized XQuery views. In *WIDM*, pages 88–91, 2002.

[Fan *et al.* 2002] Catalina Fan, John Funderburk, Hou in Lam, Jerry Kiernan, Eugene Shekita, et Jayvel Shanmugasundaram. XTABLES : Bridging Relational Technology and XML, 2002.

[Fernandez *et al.* 1998] Mary F. Fernandez, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, et Dan Suciu. Catching the Boat with Strudel : Experiences with a Web-Site Management System. In *SIGMOD Conference*, pages 414–425, 1998.

[Finance et Gardarin 1991] Béatrice Finance et Georges Gardarin. A Rule-Based Query Rewriter in an Extensible DBMS. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 248–256, Washington, DC, USA, 1991. IEEE Computer Society.

[Finance et Gardarin 1992] Béatrice Finance et Georges Gardarin. *A Rule-Based Query Optimizer with Multiple Search Strategies*. Technical report, PRiSM Laboratory University of Versailles, INRIA RODIN Project, 1992.

[Fundulaki *et al.* 2002] Irini Fundulaki, Bernd Amann, Catriel Beeri, Michel Scholl, et Anne-Marie Vercoustre. ST_YX : Connecting the XML Web to the World of Semantics. In *EDBT*, pages 759–761, 2002.

[Garcia-Molina *et al.*] Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, et Jennifer Widom. Integrating and Accessing Heterogeneous information sources in TSIMMIS. In Proceedings of the AAAI Symposium on Information Gathering, pages 61–64, Stanford, California, March 1995.

[Gardarin *et al.* 1995] Georges Gardarin, Sofiane Gannouni, Béatrice Finance, Peter Fankhauser, Wolfgang klas, Dominique Pastre, Régis Legoff, et Antonis Ramfos. IRO-DB : A Distributed System Federating Object and Relational Databases, 1995. In Object-Oriented Multibase Systems, O. Bukres and A. Elmagarmid Ed., Prentice Hall.

[Gardarin *et al.* 1996] G. Gardarin, F. Sha, et Z.-H. Tang. Calibrating the Query Optimizer Cost Model of IRO-DB. In *Proc. of the 22nd Int'l Conf. on Very Large Data Bases (VLDB)*, pages 378–389, Mumbai (Bombay), India, 1996.

[Gardarin *et al.* 2002] Georges Gardarin, Antoine Mensch, et Anthony Tomasic. An introduction to the e-xml data integration suite. In *EDBT*, pages 297–306, 2002.

[Goldfarb 1991] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, Inc., 1991.

[Graefe *et al.* 1991] Goetz Graefe, Richard L. Cole, Diane L. Davison, William J. McKenna, et Richard H. Wolniewicz. Extensible Query Optimization and Parallel Execution in Volcano. In *Query Processing for Advanced Database Systems, Dagstuhl*, pages 305–335. Morgan Kaufmann, 1991.

[Graefe et DeWitt 1987] Goetz Graefe et David J. DeWitt. The EXODUS Optimizer Generator. In *In SIGMOF Proceedings*, pages 160–172. ACM, May 1987.

[Graefe et McKenna 1991] Goetz Graefe et William J. McKenna. *The Volcano Optimizer Generator*. Computer science technical report 563, University of Colorado at Boulder, December 1991.

[Graefe et McKenna 1993] Goetz Graefe et William J. McKenna. The Volcano Optimizer Generator : Extensibility and Efficient Search. In *ICDE*, pages 209–218, 1993.

[Graefe 1987] Goetz Graefe. *Rule-Based Query Optimisation in Extensible Dabatase Systems*. PhD thesis, University of Wisconsin-Madison, November 1987.

[Graefe 1994] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1) :120–135, 1994.

[Guttag 1975] John Vogel Guttag. *The specification and application to programming of abstract data types.* PhD thesis, 1975.

[Haas *et al.* 1989] Laura Haas, Johann Freytag, Guy Lohman, et Hamid Pirahesh. Extensible Query Processing in Starburst. In *In SIGMOD Proceedings*, pages 377–388. ACM, June 1989.

[Haas *et al.* 1997] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, et Jun Yang. Optimizing queries across diverse data sources. In *VLDB*, pages 276–285, 1997.

[Halevy *et al.* 2003a] Alon Y. Halevy, Zachary G. Ives, Peter Mork, et Igor Tatarinov. Piazza : data management infrastructure for semantic web applications. In *WWW*, pages 556–567, 2003.

[Halevy *et al.* 2003b] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, et Igor Tatarinov. Schema Mediation in Peer Data Management Systems. In *ICDE*, pages 505–, 2003.

[Huck et Macherius 2000] Gerald Huck et Ingo Macherius. GMD-ISPI XQL Engine, 2000. http ://xml.darmstadt.gmd.de/xql.

[Jagadish *et al.* 2001] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, et Keith Thompson. TAX : A Tree Algebra for XML. In *DBPL*, pages 149–164, 2001.

[Kabra et DeWitt 1999] Navin Kabra et David J. DeWitt. OPT++ : an object-oriented implementation for extensible database query optimization. *VLDB Journal : Very Large Data Bases*, 8(1) :55–78, 1999.

[Kapitskaia *et al.* 1997] Olga Kapitskaia, Anthony Tomasic, et Patrick Valduriez. Dealing with Discrepancies in Wrapper Functionality. In *BDA*, pages 0–, 1997.

[Kirk *et al.* 1995] Thomas Kirk, Alon Y. Levy, Yehoshua Sagiv, et Divesh Srivastava. The Information Manifold. In C. Knoblock et A. Levy, editors, *Information Gathering from Heterogeneous, Distributed Environments*, Stanford University, Stanford, California, 1995.

[Lakshmanan *et al.* 2004] Laks V.S. Lakshmanan, Ganesh Ramesh, Hui (wendy) Wang, et Zheng (Jessica) Zhao. On Testing Satisfiability of Tree Pattern Queries. In *30th VLDB*, 2004.

[Landers et Rosenberg 1982] T. A. Landers et Ronni Rosenberg. An Overview of MULTIBASE. In *DDB*, pages 153–184, 1982.

[Lanzelotte et Valduriez 1991] Rosana S. G. Lanzelotte et Patrick Valduriez. Extending the Search Strategy in a Query Optimizer. In *VLDB*, pages 363–373, 1991.

[Levy *et al.* 1996] Alon Y. Levy, Anand Rajaraman, et Joann J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB*, pages 251–262, 1996.

[Levy 1999] Alon Y. Levy. Logic-Based Techniques in Data Integration. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

[Li *et al.* 1998] Chen Li, Ramana Yerneni, Vasilis Vassalos, Hector Garcia-Molina, Yannis Papakonstantinou, Jeffrey D. Ullman, et Murty Valiveti. Capability Based Mediation in TSIMMIS. In *SIGMOD Conference*, pages 564–566, 1998.

[Liu to appear] Tianxao Liu. *Modèle de coût pour les Tree Graph Views*. PhD thesis, Laboratoire ETIS, Université de Cergy-Pontoise, to appear.

[Lo 1992] Chi-Yuan Lo. *Query Tree Interface, Report on OODB Query Optimizer*. Technical report, Brown University, February 1992.

[Lohman 1988] Guy M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *SIGMOD Conference*, pages 18–27, 1988.

[Maitre *et al.* 1997] Jacques Le Maitre, Elisabeth Murisasco, et Monique Rolbert. From Annotated Corpora to Databases : The SgmlQL Language. In John Nerbonne, editor, *Linguistic Databases*, pages 37–58. CSLI Publications, Stanford, California, 1997.

[Manolescu *et al.* 2000] Ioana Manolescu, Daniela Florescu, Donald Kossmann, Florian Xhumari, et Dan Olteanu. Agora : Living with xml and relational. In *VLDB*, pages 623–626, 2000.

[Manolescu *et al.* 2001] Ioana Manolescu, Daniela Florescu, et Donald Kossmann. Answering xml queries on heterogeneous data sources. In *VLDB*, pages 241–250, 2001.

[Manolescu *et al.* 2004] Ioana Manolescu, Andrei Arion, Angela Bonifati, et Andrea Pugliese. Path sequence-based xml query processing. In *BDA*, 2004.

[McKenna *et al.* 1996] William J. McKenna, Louis Burger, Chi Hoang, et Melissa Truong. EROC : A toolkit for building NEATO query optimizers. In *The VLDB Journal*, pages 111–121, 1996.

[Mitchell *et al.* 1993] Gail Mitchell, Umershwar Dayal, et Stanley B. Zdonik. Control of an Extensible Query Optimizer : A Planning-Based Approach. In *19th VLDB Conference in Dublin*, 1993.

[Moro *et al.* 2005] Mirella Moura Moro, Zografoula Vagena, et Vassilis J. Tsotras. Tree-Pattern Queries on a Lightweight XML Processor, 2005.

[Naacke 1999] Hubert Naacke. *Modèles de coût pour médiateurs de bases de données hétérogènes*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, september 1999. Directeur de thèse : Georges Gardarin.

[Padawan 2005] Padawan. Proxy for All Devices Accessing the World And Neighbourhood, december 2005. ANR project, http ://padawan.ensea.fr/.

[Papakonstantinou *et al.* 1995] Yannis Papakonstantinou, Hector Garcia-Molina, et Jennifer Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260, 1995.

[Patil *et al.* 1992] Ramesh Patil, Richard F. Fikes, Peter F. Patel-Schneider, Don McKay, Tim Finin, Thomas Gruber, et Robert Neches. The DARPA Knowledge Sharing Effort : Progress Report. In Bernhard Nebel, Charles Rich, et William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning : Proceedings of the Third International Conference*, pages 777–788. Morgan Kaufmann, San Mateo, California, 1992.

[Piraresh *et al.* 1992] Hamid Piraresh, Josep M. Hellerstein, et Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD Proceedings*, pages 39–48. ACM, June 1992.

[Ramanan 2002] P. Ramanan. Efficient Algorithms for Minimizing Tree Pattern Queries. In *ACM SIGMOD*, pages 299–309, June 2002.

[Reiss 1982a] Steven P. Reiss. Eris : The Design and Implementation of an Experimental Reliational Information System. Unpublished manuscript, 1982.

[Reiss 1982b] Steven P. Reiss. *The Efficient Implementation of Flexible Relational Database Systems*. Technical report, Brown University, 1982.

[Robie *et al.* 1998] Jonathan Robie, Joe Lapp, et David Schach. XML Query Language (XQL). In *QL*, 1998.

[Rousset *et al.* 2006] Marie-Christine Rousset, P. Adjiman, Philippe Chatalic, François Goasdoué, et Laurent Simon. SomeWhere in the Semantic Web. In *SOFSEM*, pages 84–99, 2006.

[Satine 2004] Satine. Semantic-based Interoperability Infrastructure for Integrating Web Service Platforms to Peer-to-Peer Networks, 2004. projet européen IST, http ://www.srdc.metu.edu.tr/webpage/projects/satine/.

[Schwarz *et al.* 1986] Peter M. Schwarz, Walter Chang, Johann Christoph Freytag, Guy M. Lohman, John McPherson, C. Mohan, et Hamid Pirahesh. Extensibility in the Starburst Database System. pages 85–92, 1986.

[SemWeb 2004] SemWeb. Interrogation du Web Sémantique avec XQuery, 2004. projet ACI MD, http ://bat710.univ-lyon1.fr/˜semweb/.

[Shan *et al.* 1995] Ming-Chien Shan, Rafi Ahmed, Jim Davis, Weimin Du, et William Kent. Pegasus : A Heterogeneous Information Management System. In *Modern Database Systems*, pages 664–682. ACM Press and Addison-Wesley, 1995.

[Shasha *et al.* 2002] Dennis Shasha, Jason Tsong-Li Wang, et Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Symposium on Principles of Database Systems*, pages 39–52, 2002.

[Sheth et Larson 1990] A.P. Sheth et J.A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. *ACM Computing Surveys*, 22(3) :183–236, 1990.

[Sieg 1989] John Connor Sieg. *Making extensible database technology work.* PhD thesis, Boston University, 1989.

[Tomasic *et al.* 1996] Anthony Tomasic, Louiqa Raschid, et Patrick Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. In *ICDCS*, pages 449–457, 1996.

[Tomasic *et al.* 1997] Anthony Tomasic, Rémy Amouroux, Philippe Bonnet, Olga Kapitskaia, Hubert Naacke, et Louiqa Raschid. The Distributed Information Search Component (Disco) and the World Wide Web. In *SIGMOD Conference*, pages 546–548, 1997.

[Travers *et al.* 2006] Nicolas Travers, Tuyet-Tram Dang-Ngoc, et Tianxao Liu. TGV : an efficient Model for XQuery Evaluation within an Interoperable System. *International Journal of Interoperability in Business Information Systems (IBIS)*, 2, december 2006. ISSN : 1862-6378.

[W3C 1994] W3C. World Wide Web Consortium, 1994.

[W3C 1999] W3C. XML Path Language (XPath) Version 1.0, November 1999. W3C Recommendation http ://www.w3.org/TR/1999/REC-xpath-19991116.

[W3C 2004a] W3C. OWL Web Ontology Language, February 2004. Technical report, http ://www.w3.org/TR/owl-features/.

[W3C 2004b] W3C. Resource Description Framework (RDF 1.0), February 2004. Technical report, http ://www.w3.org/RDF/.

[W3C 2004c] W3C. XML Schema Part 1 : Structures, October 2004. W3C Recommendation. http ://www.w3.org/TR/xmlschema-1/.

[W3C 2006a] W3C. An XML Query Language (XQuery 1.0), June 2006. W3C Recommendation, http ://www.w3.org/TR/2006/CR-xquery-20060608/.

[W3C 2006b] W3C. XML Query Use Cases, June 2006. W3C Working Drafts, http ://www.w3.org/TR/2006/WD-xquery-use-cases-20060608/.

[W3C 2006c] W3C. XQuery Update Facility, July 2006. W3C Working Draft, http ://www.w3.org/TR/2006/WD-xqupdate-20060711/.

[WebSI 2004] WebSI. Data-Centric Web Service Integrator (WebSI), 2004. IST European Project, http ://www.ib-ia.com/websi/.

[Widom 1996] Jennifer Widom. The Starburst Active Database Rule System. *IEEE Trans. Knowl. Data Eng.*, 8(4) :583–595, 1996.

[Wiederhold 1992] G. Wiederhold. Mediators in the Architecture of Future Information System. *Computer*, 25 (3) :38–49, 1992.

[Yao et II 2004] Jingtao Yao et Ming Zhang II. A Fast Tree Pattern Matching Algorithm for XML Query. In *Web Intelligence*, pages 235–241, 2004.

Annexe A

XML

A.1 Description

Un document semi-structuré présente une forme arborescente d'éléments pouvant contenir des données multivaluées, des données textuelles, des attributs (contrairement à OEM qui ne contient pas d'attributs). Ce document est auto-descriptif contenant à la fois données et information, permettant d'identifier la structure et le sens de ces données.

XML est un méta-langage semi-structuré de représentation de données. Il a été standardisé par le *W3C* en 1998. Il est issu des langages de marquage balisés et plus particulièrement de SGML[1] ([Goldfarb 1991], [Cover *et al.* 1991], [Maitre *et al.* 1997]) son ancêtre. Ainsi XML définit comment marquer les documents.

Tout document est composé d'éléments pouvant s'imbriquer sans se chevaucher. Un élément se compose d'une balise d'ouverture (<...>), d'un contenu (texte, autres éléments) et d'une balise de fermeture (</...>).

Une balise doit contenir le nom de l'élément, il permet de l'identifier de manière unique, et ainsi de structurer le document. La balise peut contenir une série d'attributs (*nom* = "*valeur*") donnant une information supplémentaire à cet élément.

La figure 2.3 nous montre un document XML, contenant les informations décrivant trois livres de «Robin Hobb». Les éléments sont représentés en gras, les attributs en italique et le texte normal représente les données. Ainsi, ces trois livres : «*L'assassin du roi*», «*La nef du crépuscule*» et «*les secrets de castelcerf*» sont représentés par trois éléments «*book*», dans lesquelles nous pouvons identifier le titre, l'auteur et le prix du livre. Cet exemple montre la flexibilité du schéma à partir du moment où toutes les balises et attributs ne sont pas fixes, voir même absents.

[1] Standard Generalized Markup Language

```
<catalog>
  <book genres="roman" isbn="123456789">
    <title> L'assasin du roi</title>
    <author>Robin Hobb </author>
    <price currency="euros"> 16,26 </price>
  </book>
  <book genres="roman">
    <title> La nef du crépuscule </title>
    <author>Robin Hobb </author>
  </book>
  <book genres="roman">
    <title> Les secrets de Castelcerf  </title>
  </book>
</catalog>
```

Fig. A.1 – Exemple de document XML

A.2 Espace de noms

Un concept important d'XML est celui de l'**espace de noms** (ou *namespace*), qui permet de fusionner des jeux de balises dans un même document en évitant les conflits de noms. L'espace de nom introduit une notion de contexte pour une balise donnée. Dans un document XML, une balise peut avoir plusieurs significations dépendantes du contexte dans lequel il est défini. Par exemple, dans le cas d'un livre, le document contient une balise <*author*> permettant de savoir qui est l'auteur de ce livre, et dans le cadre de la Recherche, la balise *author* définit l'auteur d'un article ou d'une revue plutôt que d'un livre. On voit ainsi apparaître l'importance du contexte du document.

Les deux contextes seront distingués par deux préfixes différents associés à des *URI* permettant de préciser leurs origines (<*book :author*> ou <*research :author*>). Les préfixes doivent être définis par un attribut associé de préfixe *xmlns* au niveau d'une balise externe référençant l'*URI* de la spécification (cela peut être une DTD ou un schéma).

Ainsi, ce concept permet de partager des langages de balisages afin de pouvoir composer des documents. Un exemple est fourni dans la partie XML-schema (A.3).

A.3 Métadonnées

Les métadonnées sont une information relative à la structure des documents et aux types des données manipulées. Elles décrivent différents attributs d'informations et leur donnent signification, contexte et organisation. Deux classes de métadonnées

ont été associées au format XML : DTD et XML-schema. Les DTD ou *Document Type Definition* sont des métadonnées permettant de définir l'imbrication des éléments constituant le document. Cette structure pose toutefois quelques problèmes de typage de données, d'interprétation, de traduction et éloigné d'XML. A la suite des DTD sont apparus les XML-schema qui sont des métadonnées semi-structurées standardisés respectant les prérogatives d'XML.

Les XML-schema [W3C 2004c][Biron et Malhotra 2000] sont standardisés par le *W3C* depuis mai 2001. Malgré le fait qu'un schéma XML présente quelques complexités de formulation, il tend à remplacer les DTD, car il permet de définir un modèle beaucoup plus complet.

Un schéma définit les éléments possibles dans un document ainsi que les attributs qui leur sont associés (avec leur type de données). Ces types de données peuvent être simples ou complexes. Il existe une large variété de types simples intégrés à XML, dont les entiers, les réels, les chaînes de caractères et les dates. Les types complexes sont des types composés d'éléments assemblés à l'aide des constructeurs *sequence*, *choice* et *all* qui seront définis par l'utilisateur. Ces types complexes permettent de définir la structure de documents complexes par définition de nouveaux types et d'héritage, comme dans les langages objets.

Nous pouvons voir dans la figure A.2 que chaque élément du document XML d'origine produit un élément <xs :element name="xxx"> dans le schéma XML dont l'attribut est le nom de la balise. Les éléments complexes tels que *book* et *catalog* produisent des <xs :complexType> et des <xs :sequence> permettant d'indiquer la façon dont est bâti cet élément. Les attributs *genres, isbn, currency* produise des éléments <xs :attribute> imbriqué dans la description de l'élément concerné. Ainsi, ce XML-schema spécifie qu'un catalogue peut contenir plusieurs livres de type "*bookType*" (*max="unbound"*). Chaque livre doit contenir un élément de type titre obligatoire, tandis que l'auteur et le prix sont optionnels (*min=0, max=1* pour les références correspondantes). L'attribut *isbn* non obligatoire est défini comme un type numérique (*int*), tandis que le type du prix est de type flottant (*float* dans le type : *priceType*). Nous pouvons ainsi voir le document "*catalog*" définissant de façon exacte la structure du document ainsi que son type précis.

Les XML-schema permettent donc de représenter les métadonnées d'un document XML, exprimant sa structure exacte, de formaliser son typage. Le typage peut être défini par l'utilisateur grâce aux types complexes. De plus, la définition d'espaces de noms (*namespace*) permet de définir un contexte pour les éléments d'un document XML.

```
<xs:schema xmlns:xs="http://www.w3c.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:element name="catalog">
        <xs:complexType>
          <xs:sequence><xs:element name="book" type="bookType" min=0 max="unbound"/> </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="bookType">
        <xs:sequence>
            <xs:element ref="title" use="required"/>
            <xs:element ref="author"/>
            <xs:element name="price" type="priceType"/>
        </xs:sequence>
        <xs:attribute name="genres" use="required">
            <xs:simpleType><xs:restriction base="xs:string"/> </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="isbn">
            <xs:simpleType><xs:restriction base="xs:int"/>  </xs:simpleType>
        </xs:attribute>
    <xs/complexType>
    <xs:element name="title" min=1 max=1>
         <xs:simpleType><xs:restriction base="xs:string"/>  </xs:simpleType>
     </xs:element>
     <xs:element name="author" min=0 max=1>
         <xs:simpleType><xs:restriction base="xs:string"/> </xs:simpleType>
     </xs:element>
     <xs:complexType name="priceType">
         <xs:simpleType><xs:restriction  base="xs:float"/> </xs:simpleType>
         <xs:attribute name="currency">
              <xs:simpleType><xs:restriction  base="xs:string"/> </xs:simpleType>
         </xs:attribute>
    </xs:compleType>
</xs:schema>
```

FIG. A.2 – Exemple de XML-Schema

Annexe B

XPath

XPath [W3C 1999] est un langage permettant de se repérer dans un document XML suivant certains en-têtes. XPath considère un document comme un arbre de nœuds, et navigue ainsi à travers un document XML grâce à sa notation en chemin. Les nœuds sont divisés en trois catégories : les nœuds textuels, les éléments et les attributs. XPath permet également de sélectionner les nœud à l'aide de filtres.

Lors de son évaluation, une expression *XPath* peut renvoyer quatre sortes de résultat : un booléen, un nombre, une chaîne de caractères ou bien une collection de nœuds.

Pour chaque étape de la localisation d'un élément, l'analyse d'un chemin XPath s'effectue en trois parties :
– un axe, dirigeant le sens de recherche par rapport au nœud contextuel ;
– un nœud de test, le noeud à localiser ;
– les prédicats, sur les attributs, les positions, etc...
Les axes dirigeant le sens de recherche relie un élément à un autre par l'intermédiaire d'une association particulière. Nous pouvons distinguer 13 axes possibles entre deux nœuds :

1. **child** : association parent/enfant ;
2. **descendant** : association ancêtre/descendant ;
3. **parent** : association enfant/parent ;
4. **ancestor** : association descendant/ancêtre ;
5. **following-sibling** : enfant suivant de mêmes parents ;
6. **preceding-sibling** : enfant précédent de mêmes parents ;
7. **following** : tous les nœuds suivant le nœud courant ;
8. **preceding** : tous les nœuds précédents le nœud courant ;
9. **attribute** : association nœud/attribut ;
10. **namespace** : tous les nœuds correspondant au contexte d'espace de noms du nœud ;

11. **self** : le nœud selectionné ;
12. **descendant-or-self** : association ancêtre/descendant, le nœud courant est inclu ;
13. **ancestor-or-self** : association descendant/ancêtre, le nœud courant est inclu.

Ces différents axes permettent de définir les types de parcours d'arbre possible en *XPath*. Ainsi, dans le cadre des *TGV*, les liens de nœuds doivent pouvoir contenir ces associations. Certaines de ces associations sont abrégées et seront utilisées en tant que tel : *child* "/", *descendant* "//", *parent* "..", *attribute* "@", *namespace* "* :", *self* ".". La syntaxe exacte de l'expression d'un *XPath* correspondant à la sélection d'un élément fils utilise la syntaxe *child : :*. La syntaxe simplifiée équivalente à la sélection de l'élément fils correspond au nom de l'élément, ainsi une expression *XPath* peut être écrite de manière plus légère et plus intuitive. Pour des raisons de commodité les chemins que nous utilisons ne sont que sous la forme abrégée (si possible).

La première phase de localisation indique le sens de recherche par rapport au nœud contextuel. Les axes les plus courants sont *child* pour indiquer tous les enfants directs du nœud contextuel, *descendant* pour tous les descendants du nœud contextuel. Pour l'axe de recherche spécifié, une expression *XPath* sélectionne les nœuds correspondants au nœud de test. Il contient des éléments, qui peuvent être de trois types : *attribute* pour les attributs, *namespace* pour les espaces de noms et *element* pour tous les autres. Une fois le type de nœud sélectionné, la dernière phase s'occupe des prédicats. L'expression *XPath* peut alors renvoyer le résultat de la requête.

Les filtres dans une expression *XPath* sont des prédicats associés à l'élément courant. Ces filtres permettent de restreindre l'ensemble des documents sélectionnés par le prédicat qui lui est associé. Un filtre se défini à l'aide de crochets "[" "]", le prédicat est alors présent entre les crochets.

Certaines fonctions sont utilisées dans les expressions chemins pour retirer des informations particulières. les appels de fonction permettent de récupérer une structure particulière à partir du nœud contextuel (nœud, ensemble de nœuds, texte, booléen, entier). Une bibliothèque des fonctions accessible est disponible sur le site du [W3C 1999].

Le langage d'expression *XPath* permet donc d'adresser simplement et efficacement tout chemin d'un document XML. XPath est aussi le langage utilisé par XQuery, langage permettant de formuler des requêtes plus élaborées, que nous présentons en annexe C.

Annexe C

XQuery

XQuery, créé le 15 février 2001, est le langage aujourd'hui utilisé pour l'interrogation de documents XML. Il est issu de Quilt [Chamberlin *et al.* 2000] et reprend les avantages de XQL [Huck et Macherius 2000], XML-QL [Deutsch *et al.* 1998] et XPath [W3C 1999]. La version étudiée ici est celle de juin 2006.

C.1 Grammaire

Les requêtes *XQuery* sont composées d'expressions nommées **FLWOR** (prononcé "flower") en raison des mots-clés qui les composent : *for, let, where, order by, return.* Ces expressions peuvent être utilisées pour mettre en commun les données de plusieurs documents et structure le résultat en un nouveau document XML. La grammaire complète du langage de requête *XQuery* disponible sur le *Web* [W3C 2006a] est complexe, dans le cadre de notre travail, nous simplifierons notre grammaire à celle présente ci-dessous. Par rapport à la grammaire présentée sur le site du W3C, celle que nous utilisons ne contient pas la notion de typage.

Nous allons étudier en détail chacune des caractéristiques de cette grammaire pour nous permettre de les utiliser dans nos travaux sur *XQuery*. Ainsi, nous verrons les clauses *let et for*, puis la clause *where*, la clause *return*, la clause *order by* et la clause *If/Then/Else.*

Clauses *let* et *for*
Le but des clauses *let* et *for* dans une expression FLWOR est de produire un ensemble d'arbres où chaque arbre est composé d'une ou plusieurs variables. Chaque variable est associée à une expression.

Dans une clause *for*, chaque expression est analysée et la variable prend *chacune* des valeurs de l'expression l'une après l'autre. L'exemple suivant est une partie de requête dont la clause *for* ne contient qu'une variable. La variable $f prendra tour à

XQuery	: :=	(*Function*)* Expr | "<"string">" (*Expr*)+ "</"string">" ;
Expr	: :=	*FLWOR* | *Path* | *XPath* | *IfClause*
		| "(" *Expr SetOperator Expr* ")" ;
FLWOR	: :=	(*ForClause* | *LetClause*)+
		("where" *WhereClause*) ? (*OrderClause*) ?
		"return " *ReturnClause* ;
ForClause	: :=	"for" *Var*(, *Var*)* ;
LetClause	: :=	"let" "$" string " : :=" *Expr* (, "$"string " : :=" *Expr*)* ;
WhereClause	: :=	*Predicate* (("and" | "or") *Predicate*)* ;
OrderClause	: :=	"order by" *Path* (, *Path*)*
ReturnClause	: :=	("{" *Expr* "}")
		| ("<"string">" (*ReturnClause*)+ "</"string">") ;
IfClause	: :=	"if" *Predicate* "then" *Expr* "else" *Expr* ;
Var	: :=	"$"string in " *Declaration* ;
Declaration	: :=	("collection" | "document") "("'string"')" (*XPath*) ? | *Expr* ;
Path	: :=	"$"string*XPath* (EndXPath) ? ;
Predicate	: :=	*Val Comp Val* | 'string' "(" ((*Val* ",")* *Val*) ? ")"
		| (("some" | "every") *Var* "in" *Expr* "satisfies" (*Expr* | *Predicate*) ;
Comp	: :=	">" | "<" | "=" | "<=" | ">=" | "! =" ;
Val	: :=	'string' | Number | Expr ;
XPath	: :=	(*Axis* | "(" *XPath* (*SetOperator XPath*) ? ")")+ ;
Axis	: :=	"/" (string " : :") ? *Element* | "//" *Element* ;
SetOperator	: :=	("|" | "union") | "intersect" | "except" ;
EndXPath	: :=	"/" (*Attribut* | *Element* | "text()") ;
Element	: :=	(*QName* | "." | "..") ("[" *Predicate* "]") ? ;
Attribute	: :=	"@" *QName* ;
QName	: :=	(string (" :") ? " :") ? string ;
Function	: :=	"declare function" QName "(" "$"string"as" "element"
		("," "$"string"as" "element")* ")" "as" "element"
		"{" *Expr* "}" ;

Tab. C.1 – Grammaire XQuery non typée

tour toutes les instances *book* de la collection *catalog*.

```
for $f in collection("catalog")/book
```

Dans la clause *let*, l'expression associée à chaque variable est analysée et la variable prend la valeur du résultat de l'expression. L'exemple suivant est une partie de requête dont la clause *let* ne contient une déclaration sous forme d'une clause *for*. La variable $l prendra la valeur résultant de la concaténation de toutes les instances *book* de la collection *catalog*.

```
let $l := for $b in collection("catalog")/book
```

Une expression *FLWR* peut comporter plusieurs clauses *for* et *let* imbriquées ou non. Les expressions de ces clauses peuvent utiliser des variables des clauses précédentes.

Clause *where*

Le but de la clause optionnelle *where* est de filtrer l'ensemble d'arbres généré par les clauses *for* et *let*. L'expression de la clause *where* est évaluée une fois pour chaque

arbre. Si le résultat de cette évaluation est vrai, alors l'arbre est conservé et utilisé par la clause *return*. Sinon, l'arbre est éliminé.

L'exemple suivant montre l'utilisation de la clause *where* sur notre collection *catalog*. Dans la clause *where*, les arbres vérifient le prédicat *$f/price < 15*, ce qui veut dire que la requête ne retournera que les livres de moins de 15 euros.

```
for $f in collection("catalog")/book
where $f/price < 15
return $f
```

Clause *return*
La clause *return* produit un arbre XML pour chacune des données du flot de sortie. Ce résultat est donné par la concaténation des balises définies dans cette clause et des valeurs requises par les XPath ou les requêtes imbriquées. Si aucune clause *order by* n'a été spécifiée dans la requête, l'ordre du flot d'arbres est déterminé par l'ordre des séquences retournées par la clause *for*. Sinon, l'ordre est déterminé par la clause *order by* (cf. C.1).

La clause *return* construit un nouveau document XML à partir du flot d'arbres créés dans la requête. Il est possible d'ajouter des balises dans le résultat. Dans l'exemple ci-dessous, la clause *return* créer un ensemble d'arbres dont la racine est *titre* et le contenu est le texte contenu dans l'élément *title*.

```
for $f in collection("catalog")/book
return <title> {$f/title/text()} </title>
```

Clause *Order By*
Le but de la clause optionnelle *order by* est de trier le résultat de la requête suivant un ou plusieurs critères. Pour chaque arbre du flot de sortie, les critères sont évalués sur les variables contenues dans cet arbre. L'ordre de deux arbres est déterminé en comparant, de gauche à droite, les valeurs de leurs évaluations. La clause *order by* peut trier le résultat d'une expression FLWOR même si le critère de tri n'est pas présent dans le résultat. L'exemple ci-dessous illustre l'ordonnancement. Dans la clause *order by*, les livres sont ordonnés selon le prix, puis on retourne le titre de chaque livre.

```
for $f in collection("catalog")/book
order by $f/price
return $f/title
```

Clause *If/Then/Else*
La clause conditionnelle *If/Then/Else* permet de donner une alternatif d'évaluation en fonction de l'expression comparée. En effet, cette clause est décomposée en trois parties :
- *If* : permet de faire le test sur l'expression contenue ;
- *Then* : Cette expression est évaluée si *If* retourne *vrai* ;
- *Else* : Cette expression est évaluée si *If* retourne *faux*.

```
for $f in collection("catalog")/book
return <title>
      {if ($f/price < 15) then $f/title/text() else "too expensive"}
</title>
```

La requête ci-dessus recherche dans l'ensemble des livres existants, si le prix de celui-ci est inférieur à 15, le titre est retourné, sinon le mot "too expensive" est retourné.

C.2 Optionalité

Le langage *XQuery* spécifie la notion d'optionalité des résultats. En effet, les *XPath* demandés dans la clause *return* ne sont pas obligatoires dans le résultat, contrairement aux *XPath* requis dans les clauses *for*, *let* et *where*. Cette notion importante nous permet de voir qu'un document XML résultant d'une requête *XQuery* ne contient que les informations nécessaires, l'absence d'une information dans la clause *return* ne doit pas être discriminante lors de la construction d'un résultat.

```
for $f in collection("catalog")/book
where exists ($f/title)
return <books>
   {$f/title}
   { for $g in $f/section return $g/p }
</books>
```

Dans la requête ci-dessus, nous pouvons voir que le *titre* d'un catalogue est obligatoire (*exists ($f/title)*) et apparaîtra donc dans la clause *return*. Par contre, la requête imbriquée $g est optionnelle par rapport à la requête $f. Ainsi, les paragraphes des sections d'un livre seront optionnelles. En effet, quand bien même un livre ne contient pas de paragraphes, son titre doit apparaître dans l'ensemble résultat.

Annexe D

Abstract Data Types : TGV

Cette annexe récapitule tous les types abstraits des éléments du TGV décrit dans le chapitre 3.

D.1 XPath

TA 1 Element E			
Utilise : string, boolean **Opérations**			
O1	createElement	string × boolean	→ E
O2	labelElement	E	→ string
O3	isChild	E	→ boolean

TA 2 XPath XP			
Utilise : E, boolean			
Opération			
O1	createXPath		→ XP
O2	first	XP	→ E
O3	last	XP	→ E
O4	addLast	XP × E	→ XP
O5	deleteFirst	XP	→ XP
O6	isEmpty	XP	→ boolean
O7	isAllChildren	XP	→ boolean
Pré-conditions : $xp \epsilon XP$;			
P1	define (first (xp))	⟺ isEmpty (xp) = false	
P2	define (last (xp))	⟺ isEmpty (xp) = false	
P3	define (deleteFirst (xp))	⟺ isEmpty (xp) = false	

D.2 TreePattern

TA 3 Node N

Utilise : string

Opérations

O1	createNode	string	→ N
O2	label	N	→ string
O3	isRoot	N	→ boolean
O4	getNodeLink	N	→ NL
O5	getNextNodeLink	N × int	→ NL
O6	getSizeNodeLink	N	→ int
O7	getConstraintLink	N × int	→ CL
O8	getSizeConstraintLink	N	→ int
O9	getTreePattern	N	→ TP

TA 4 Axis A

Utilise : String ϵ {'ancestor', 'ancestor-or-self', 'child', 'descendant', 'descendant-or-self', 'following', 'following-sibling', 'preceding', 'preceding-sibling'}

Opérations

O1	createAxis	String	→ A
O2	getAxis	A	→ String

TA 5 NodeLink NL

Utilise : N, Axis, int, boolean

Opérations

O1	createNodeLink	N × N × int × Axis × boolean	→ NL
O2	getPosition	NL	→ int
O3	getLinkAssociation	NL	→ Axis
O4	isMandatory	NL	→ boolean
O5	getPreviousNode	NL	→ N
O6	getNextNode	NL	→ N

Pré-conditions : $n \epsilon N$; $i \epsilon int$;

P1	define (getNodeLink (n))	⟺ ¬ isRoot (n)
P2	define (getNextNodeLink (n, i))	⟺ i >= 0 && i < getSizeNodeLink (n)

TA 6 TreePattern TP

Utilise : string, XP, N, int, boolean

Operations

O1	createTP	string × XP	→ TP
O2	getDeclarationName	TP	→ string
O3	getRoot	TP	→ N
O4	getSource	TP	→ N
O5	getNode	TP × XP	→ N
O6	addChildNode	N × string × int × boolean × boolean	→ N
O7	addXPathToNode	N × XP × boolean	→ N
O8	addXPathToTreePattern	TP × XP × boolean	→ TP
O9	addConstraintToTreePattern	TP × XP × boolean × C	→ TP

Pré-conditions : $n \epsilon N$; $xp \epsilon XP$; $b \epsilon boolean$;

P1	define (addXPathToNode (n, xp, b))	⟺ isEmpty (xp) = false

TA 7 SourceTreePattern extends TreePattern STP

Utilise : string, XP, P

Operations

O1	createSTP	string × string × XP	→ STP
O2	getVariableSTP	STP	→ string
O3	getSourceSTP	STP	→ N

TA 8 ReturnTreePattern extends TreePattern RTP

Utilise : string, XP

Operations

O1	createRTP	string	→ RTP
O2	addXPathToRTP	RTP × XP	→ RTP
O3	addConstraintHyperlinkToRTP	RTP × CH	→ RTP
O4	getConstraintHyperlink	RTP	→ CH

Pré-conditions : $rtp \in RTP$; $xp \in XP$;

P1 define (addXPathToRTP (rtp, xp)) $\Longleftrightarrow$ isAllChildren (xp) = true

TA 9 IntermediateTreePattern extends TreePattern ITP

Utilise : string

Opérations

O1	createITP	string × string	→ ITP

TA 10 AggregateTreePattern extends ReturnTreePattern ATP

Utilise : string, C

Opérations :

O1	createATP	string × C	→ ATP
O2	getATPConstraint	ATP	→ C

D.3 Constraint

TA 11 Constraint C

Utilise : string

Opérations

O1	createConstraint	string	→ C
O2	getConstraintName	C	→ string
O3	getConstraintLink	C	→ CL

TA 12 Predicate P extends Constraint

Utilise : string

Opérations

O1	createPredicate	string × string	→ P
O2	getComparator	P	→ string
O3	getConstant	P	→ string

TA 13 ListParameter LP

Utilise : string, int

Operations

O1	createList		→ LP
O2	listSize	LP	→ int
O3	addLast	LP × string	→ LP
O4	getParam	LP × int	→ string

Pré-conditions : $s \in string$; $i \in int$; $lp \in LP$;

P1 define (getParam (lp, i)) $\Longleftrightarrow$ i > 0 && $\Longleftrightarrow$ i <= listSize (lp)

TA 14 Function F extends Constraint

Utilise : string, ListString

Opérations

O1	createFunction	string × ListParameter	→ F
O2	getFunction	F	→ string
O3	getParameters	F	→ ListParameter

TA 15 ConstraintLink CL

Utilise : Node N, Constraint C, TreePattern TP, E ϵ {N, C, CL, TP}, Constraint C, XP, int, boolean

Opérations

O1	createConstraintLink	E × C × i	→ CL
O2	getConstraint	CL	→ E
O3	getConstraintElement	CL	→ N
O4	getConstraintNumber	CL	→ int

Pré-conditions : $n \epsilon N$; $s \epsilon string$; $i \epsilon int$; $c \epsilon C$;

P1	define (getConstraintLink (n, i))	⟺ i > 0 && i <= getMaxConstraintLink (e)
P2	define (getConstraintLink (n, i))	⟺ getConstraint (createConstraintLink (n, c, i)) = c

D.4 Hyperlink

TA 16 JoinHyperlink JH extends Hyperlink

Utilise : Constraint, E ϵ {Node, ConstraintLink, Constraint}

Opérations

O1	createJoinHyperlink	E × E × C	→ HJ
O2	leftJH	HJ	→ E
O3	rightJH	HJ	→ E
O4	getConstraintJH	HJ	→ C

TA 17 ConstraintHyperlink CH extends Hyperlink

Utilise : E ϵ {C, CH}, C, RTP, boolean

Opérations

O1	createConstraintHyperlink	E × E × boolean	→ CH
O2	leftCH	CH	→ E
O3	rightCH	CH	→ E
O4	getConnectorCH	CH	→ boolean
O5	getConstraintHyperlinkRTP	CH	→ RTP

Pré-conditions : $rtp \epsilon RTP$; $ch \epsilon CH$;

P1	define (getConstraintHyperlink (rtp)) ⟺ addConstraintHyperlinkToRTP (rtp, ch)
P2	define (getConstraintHyperlinkRTP (ch)) ⟺ addConstraintHyperlinkToRTP (rtp, ch)

TA 18 DirectionalHyperlink DH extends Hyperlink

Utilise : E ϵ {Node N, TreePattern TP}, boolean

Opérations

O1	createDirectionalHyperlink	E × E × boolean	→ DH
O2	fromDH	DH	→ E
O3	toDH	DH	→ E
O4	optionalPH	DH	→ boolean

Pré-conditions : $\forall\ dh_1, dh_2 \epsilon DH$;

P1	toDH (dh_1) != toDH (dh_2)

TA 19 ExplorationHyperlink EH extends DirectionalHyperlink

Utilise : Node N, TreePattern TP, boolean

Opérations

O1	createExplorationHyperlink	N × TP × boolean	→ EH

TA 20 GeneralizedHyperlink GH extends DirectionalHyperlink

Utilise : Node N, TreePattern TP, boolean

Opérations

O1	createGeneralizedHyperlink	TP × N × boolean	→ GH

TA 21 SetHyperlink SH extends Hyperlink

Utilise : Node N, TreePattern TP, Constraint C, String, int

Opérations

O1	createSetHyperlink	(TP) × N × C	→ SH
O2	getTPSH	SH × int	→ TP
O3	getMaxTPSH	SH	→ int
O4	resultSH	SH	→ N
O5	getConstraintSH	SH	→ C

Pré-conditions : $l \in (TP)$; $i \in int$;

P1	define (getTPSH (l, i))	⇔ i > 0 && i <= getMaxTPSH (l)

TA 22 IfThenElseHyperlink IH extends Hyperlink

Utilise : Node N, Constraint C, E $\in$ {N, TP}

Opérations

O1	createIfThenElseHyperlink	C × E × E × N	→ IH
O2	ifIH	IH	→ C
O3	thenIH	IH	→ E
O4	elseIH	IH	→ E
O5	resultIH	IH	→ N

D.5 TGV

TA 23 TreeGraphView TGV

Utilise : string, Node N, XPath XP, Hyperlink HY, TreePattern TP

Opérations

O1	createTGV		→ TGV
O2	addTreePattern	TGV × TP	→ TGV
O3	getTreePattern	TGV × string	→ TP
O4	getTGVNode	TGV × XP	→ N
O5	addXPathToTGV	TGV × XP × boolean	→ TGV
O6	addConstraintToTGV	TGV × XP × boolean × C	→ TGV
O7	addPH	TGV × XP × XP × boolean	→ TGV
O8	addEH	TGV × XP × string × boolean	→ TGV
O9	addGH	TGV × string × XP × boolean	→ TGV
O10	addJH	TGV × XP × boolean × XP × boolean × C	→ TGV
O11	addSH	TGV × string × string × XP × string	→ TGV
O12	addIH	TGV × XP × C × string × string × XP	→ TGV
O13	addCH	TGV × C × C × boolean	→ TGV

D.6 Annotations

TA 24 Information I

Utilise : String

Opérations

O1	createInformation	String	→ I
O2	getInformation	I	→ String

TA 25 ListTGVObject LT

Utilise : Element E ϵ {Node, NodeLink, Constraint, ConstraintLink, Hyperlink, TreePattern, TGV}

O1	createListTGVObject		→ LT
O2	firstLT	LT	→ E
O3	lastLT	LT	→ E
O4	addLastLT	LT × E	→ LT
O5	deleteFirstLT	LT	→ LT
O6	isEmptyLT	LT	→ boolean

Pré-conditions : *lt* ϵ *LT*;

P1	define (firstLT (lt))	⟺ isEmptyLT (lt) = false
P2	define (lastLT (lt))	⟺ isEmptyLT (lt) = false
P3	define (deleteFirstLT (lt))	⟺ isEmptyLT (lt) = false

TA 26 Annotation A

Utilise : String, LT, I

Opérations

O1	createAnnotation	LT × I	→ A
O2	getAnnotation	A	→ I
O2	getListTGVObject	A	→ LT

TA 27 AnnotationSet AS

Utilise : A

O1	createAnnotationSet		→ AS
O2	firstAS	AS	→ A
O3	lastAS	AS	→ A
O4	addLastAS	AS × A	→ AS
O5	deleteFirstAS	AS	→ AS
O6	sizeAS	AS	→ int

Pré-conditions : *l* ϵ *AS*; *i* ϵ *int*;

P1	define (getAS (l, i))	⟺ i >= 0 && i < sizeAS (AS)
P2	define (deleteAS (l, i))	⟺ i >= 0 && i < sizeAS (AS)

Annexe E

Compléments d'Algorithmes pour TreePattern

Afin d'éclaircir les opérations du *TA TreePattern*, nous allons étudier les algorithmes de récupération d'un nœud grâce à un *XPath* (*O5*) et celui d'ajout d'un nœud grâce à un *XPath* (*O6*, *O7* et *O8*).

La fonction **getNode** (*O5*) retourne le nœud spécifié par un *XPath xp*, dans le *TreePattern tp*. Il faut pour cela parcourir les différents éléments de *xp* tout en parcourant l'arbre de *tp* en sélectionnant les nœuds dont le label est identique à l'élément en cours. Si le nœud n'existe pas, cela équivaut à dire que *xp* n'existe pas dans *tp*.

Algorithme 1 getNode (tp, xp)

```
n := getRoot (tp)
while !isEmpty (xp) do
    m := null
    for all i de 0 à getSizeNodeLink (n) do
        nl := getNodeLink (n, i)
        if isDirectSon (nl) == isChild (first (xp))
        && label (getChildNode (nl)) == labelElement (first (xp)) then
            m := getChildNode (nl)
        end if
    end for
    if m == NULL then
        return NULL
    else
        n := m
    end if
    xp := deleteFirst (xp)
end while
return n
```

Cet algorithme nous permet donc de retrouver le nœud correspondant à un *XPath*.

Pour cela, il nous faut commencer par la racine du *TreePattern* concerné (ligne 1). Ensuite, pour chacun des éléments de *xp* (ligne 2), nous devons retrouver le nœud correspondant. Pour se faire nous parcourons l'ensemble des *NodeLink* du nœud courant (ligne 4 et 5), et vérifier si le lien d'ascendance du *NodeLink* est identique à celui du nœud et si le label de celui-ci et de l'élément son identiques (ligne 6). Ensuite, si l'élément courant n'a pas été trouvé, nous arrêtons l'algorithme (ligne 10 et 11). Sinon, nous recommençons le processus pour le nœud qui a été sélectionné (ligne 13) et l'élément suivant (ligne 15). Une fois que le dernier élément de l'*XPath* a été trouvé dans le *TreePattern* nous pouvons renvoyer le nœud correspondant (ligne16).

La fonction **addXPathToTreePattern** donne l'ajout d'un *XPath* dans un *TreePattern* à l'aide des opérations *addXPathToNode* et *addChildNode*.

Algorithme 2 addXPathToTreePattern (tp, xp, m)

```
 1: p := getRoot (tp)
 2: xp1 := createXPath ()
 3: while !isEmpy (xp) do
 4:    e := first (xp)
 5:    xp1 := addLast (xp1, e)
 6:    xp := deleteFirst (xp)
 7:    n1 := getNode (tp, xp1)
 8:    if n1 == null then
 9:       k := getSizeNodeLink (n) + 1
10:       n1 := addChildNode (n1, labelElement (e), k, m, isChild (e))
11:    end if
12:    n := n1
13: end while
14: return tp
```

De la même manière que pour *getXPath*, nous devons ajouter *xp* à partir de la racine (ligne 1). Pour chaque élément de *xp* nous devons vérifier qu'il n'existe pas un nœud correspondant, et dans ce cas, lui associer le reste du *XPath*. Pour ce faire, nous parcourons *xp* (ligne 3), et récupérons le premier élément (ligne 4). Nous avons besoin d'un *XPath* temporaire pour récupérer un à un les nœuds du *TreePattern*, pour cela nous ajoutons l'élément courant à un nouvel *XPath* (ligne 5). Puis, nous vérifions s'il existe un nœud pour ce nouvel *XPath* (ligne 7). Si *p1* n'est pas un nœud, alors il nous faut créer le nœud correspondant à l'élément courant (ligne 10), le numéro du lien (ligne 9) correspond au nombre de *NodeLink* associés au nœud courant. Nous continuons alors le processus jusqu'à ce *xp* soit vide (ligne 3), donc qu'il ne reste nlus de nœuds à créer. Une fois le processus terminer, nous pouvons retourner le *TreePattern* courant (ligne 14).

Annexe F

Cas d'usage

Nous énumérons ici chaque cas d'usage du [W3C 2006b]. Nous détaillons chacune des requêtes du cas d'usage XMP, et nous donnons la requête la plus caractéristique pour les autres cas d'usage. Pour chacun d'eux, la requête XQuery est fournise, associée à un TGV.

F.1 Use Cases XMP

Query 1 : *List books published by Addison-Wesley after 1991, including their year and title*

```
<bib> {
    for $b in doc("http ://bstore1.example.com/bib.xml")/bib/book
    where $b/publisher = "Addison-Wesley" and $b/@year > 1991
    return <book year="{ $b/@year }"> { $b/title } </book>
} </bib>
```

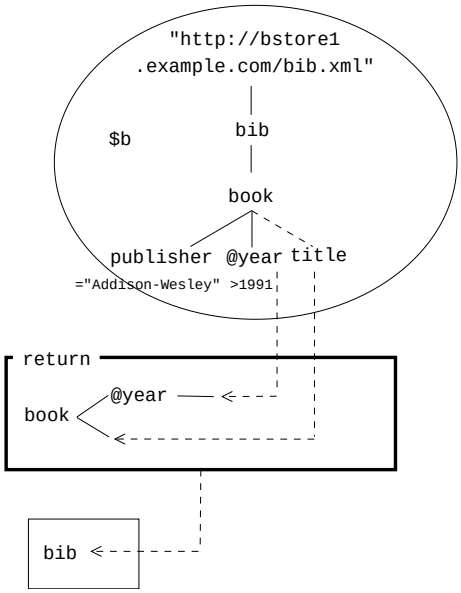


Fig. F.1 – TGV for query XMP 1

Query 2 : *Create a flat list of all the title-author pairs, with each pair enclosed in a "result" element*

```
<results>
{
        for $b in doc("http ://bstore1.example.com/bib.xml")/bib/book,
                $t in $b/title,
                $a in $b/author
        return <result> { $t } { $a } </result>
}
</results>
```

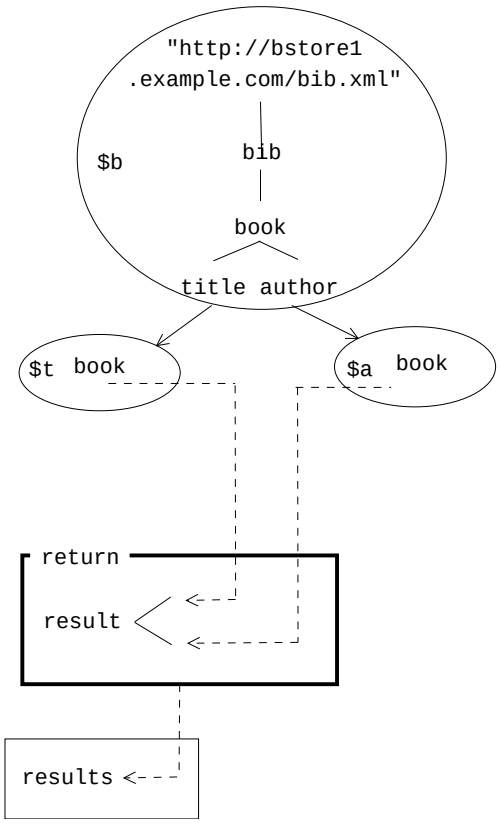


Fig. F.2 – TGV for query XMP 2

Query 3 : *For each book in the bibliography, list the title and authors, grouped inside a "result" element*

```
<results>
{
        for $b in doc("http ://bstore1.example.com/bib.xml")/bib/book
        return <result> { $b/title } { $b/author } </result>
}
</results>
```

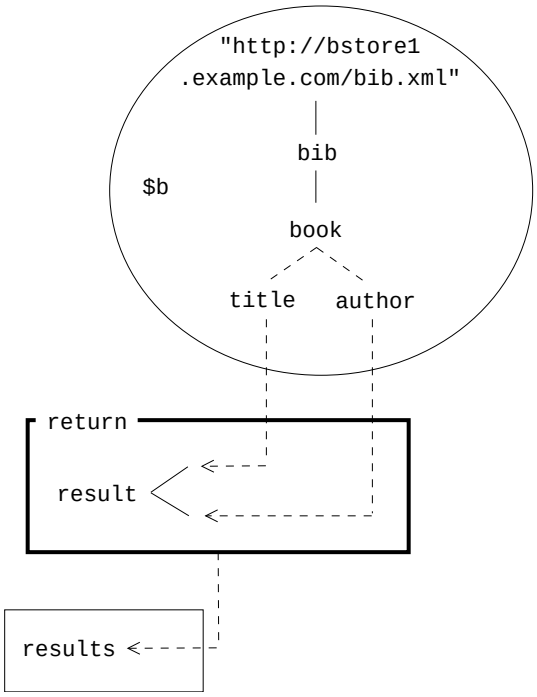


Fig. F.3 – TGV for query XMP 3

Query 4 : *For each author in the bibliography, list the author's name and the titles of all books by that author, grouped inside a "result" element*

```
<results>
        {let $a := doc("http ://bstore1.example.com/bib/bib.xml")//author
        for $last in distinct-values($a/last),
                $first in distinct-values($a[last=$last]/first)
        order by $last, $first
        return
                <result>
                        <author>
                                <last>{ $last }</last> <first>{ $first }</first>
                        </author>
                        {
                        for $b in doc("http ://bstore1.example.com/bib.xml")/bib/book
                        where some $ba in $b/author
                                satisfies ($ba/last = $last and $ba/first=$first)
                        return $b/title
                        }
                </result>
        }
</results>
```

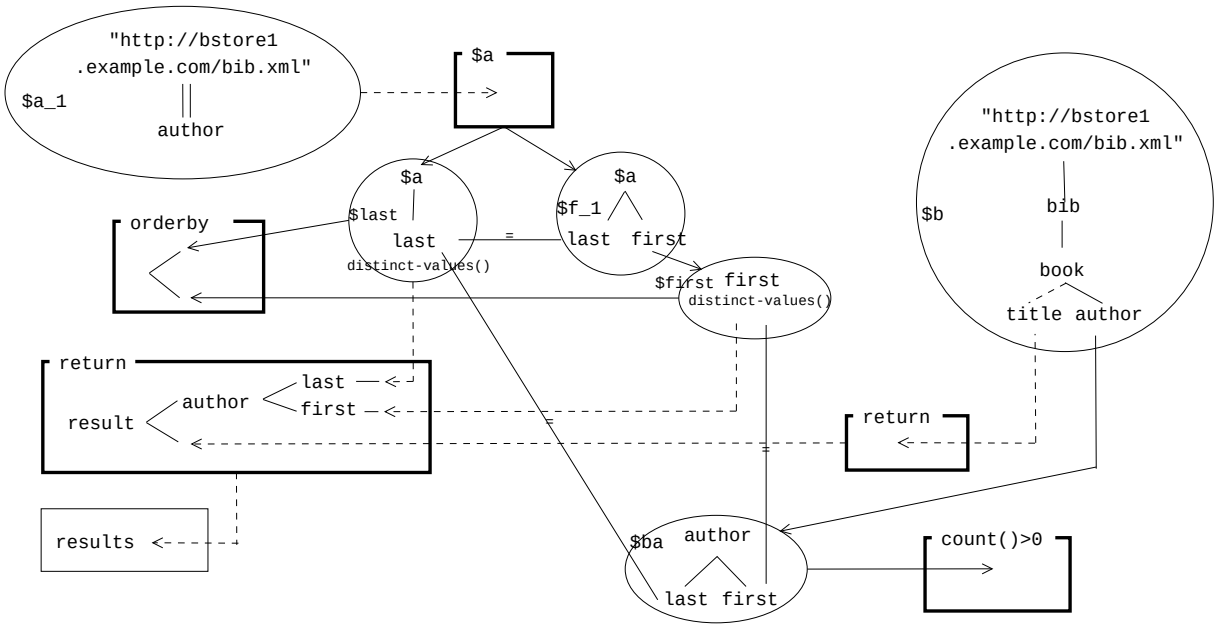


Fig. F.4 – TGV for query XMP 4

Query 5 : *For each book found at both bstore1.example.com and bstore2.example.com, list the title of the book and its price from each source*

```
<books-with-prices>
{
        for $b in doc("http ://bstore1.example.com/bib.xml")//book,
                $a in doc("http ://bstore2.example.com/reviews.xml")//entry
        where $b/title = $a/title
        return
                <book-with-prices>
                        { $b/title }
                        <price-bstore2>{ $a/price/text() }</price-bstore2>
                        <price-bstore1>{ $b/price/text() }</price-bstore1>
                </book-with-prices>
}
</books-with-prices>
```

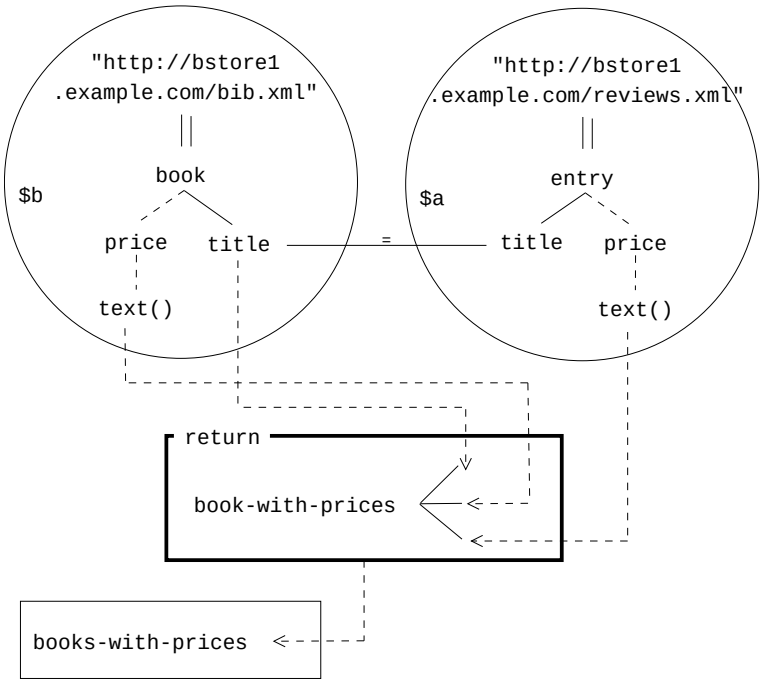


Fig. F.5 – TGV for query XMP 5

Query 6 : *For each book that has at least one author, list the title and first two authors, and an empty "et-al" element if the book has additional authors*

```
<bib>
{
        for $b in doc("http ://bstore1.example.com/bib.xml")//book
        where count($b/author) > 0
        return
                <book>
                        { $b/title }
                        {
                                for $a in $b/author[position()<=2]
                                return $a
                        }
                        {
                                if (count($b/author) > 2)
                                then <et-al/>
                                else ()
                        }
                </book>
}
</bib>
```

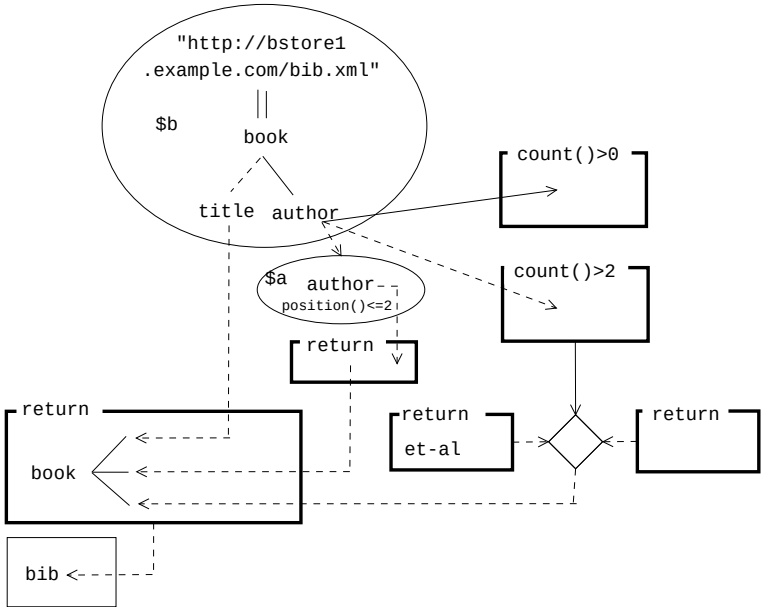


FIG. F.6 – TGV for query XMP 6

Query 7 : *List the titles and years of all books published by Addison-Wesley after 1991, in alphabetic order*

```
<bib>
{
    for $b in doc("http ://bstore1.example.com/bib.xml")//book
    where
        $b/publisher = "Addison-Wesley"
        and $b/@year > 1991
    order by $b/title
    return
        <book>
            { $b/@year }
            { $b/title }
        </book>
}
</bib>
```

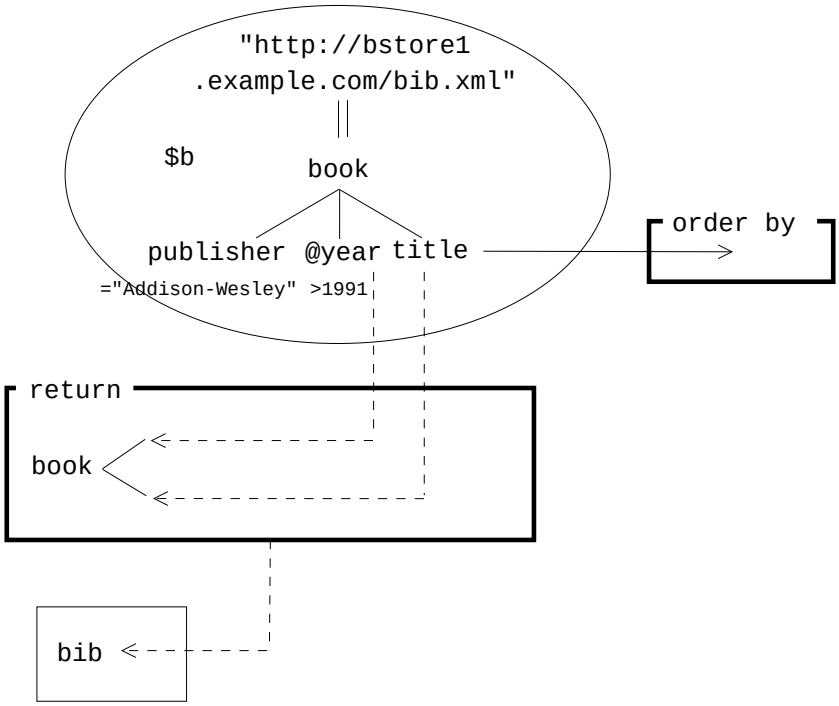


Fig. F.7 – TGV for query XMP 7

Query 8 : *Find books in which the name of some element ends with the string "or" and the same element contains the string "Suciu" somewhere in its content. For each such book, return the title and the qualifying element*

```
for $b in doc("http ://bstore1.example.com/bib.xml")//book
let $e := $b/*[contains(string(.), "Suciu")
                    and ends-with(local-name(.), "or")]
where exists($e)
return
    <book>
        { $b/title }
        { $e }
    </book>
```

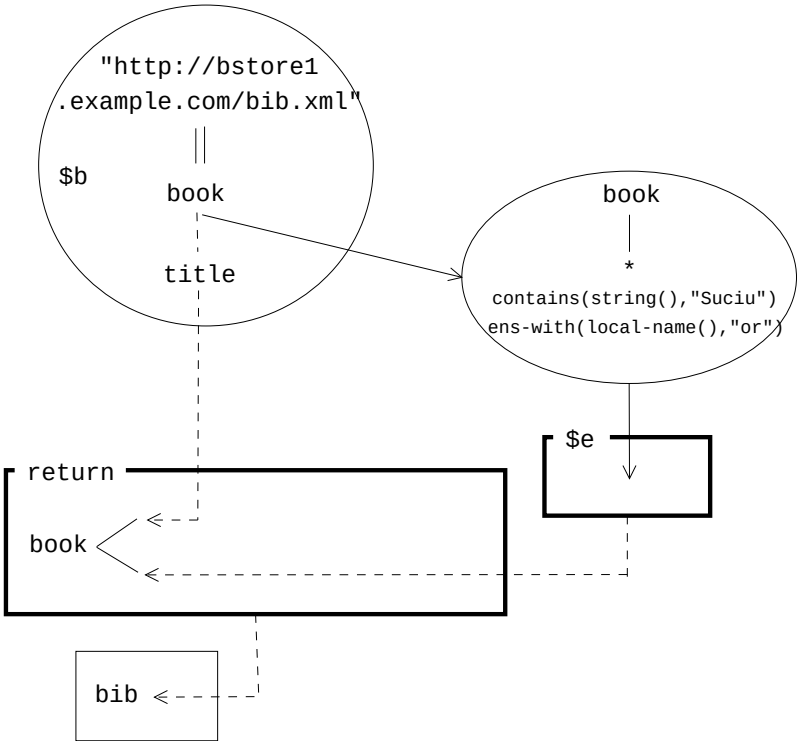


Fig. F.8 – TGV for query XMP 8

Query 9 : *In the document "books.xml", find all section or chapter titles that contain the word "XML", regardless of the level of nesting*

```
<results>
{
        for $t in doc("books.xml")//(chapter | section)/title
        where contains($t/text(), "XML")
        return $t
}
</results>
```

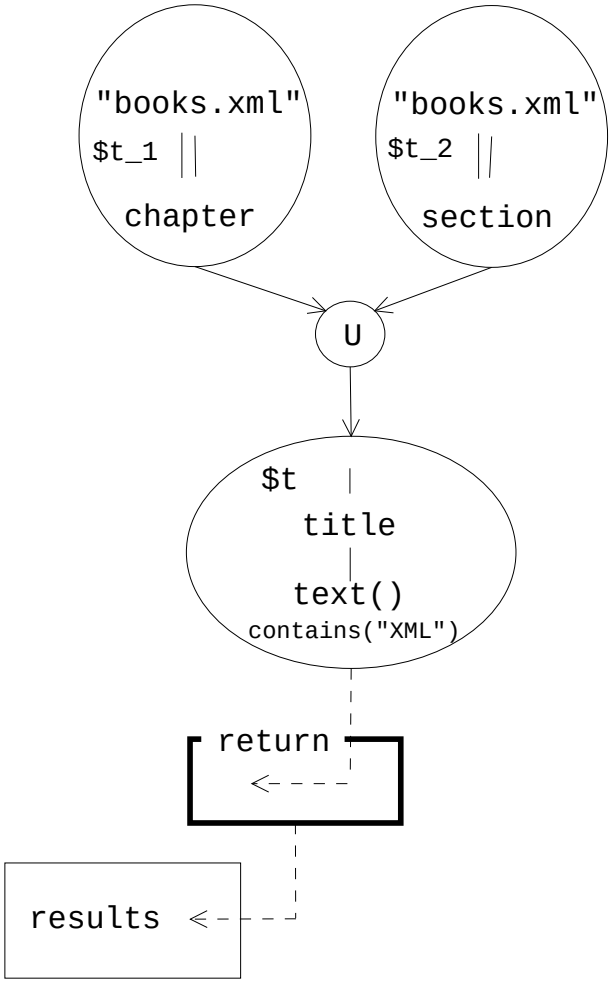


Fig. F.9 – TGV for query XMP 9

Query 10 : *In the document "prices.xml", find the minimum price for each book, in the form of a "minprice" element with the book title as its title attribute*

```
<results>
{
        let $doc := doc("prices.xml")
        for $t in distinct-values($doc//book/title)
        let $p := $doc//book[title = $t]/price
        return
                <minprice title="{ $t }">
                        <price>{ min($p) }</price>
                </minprice>
}
</results>
```

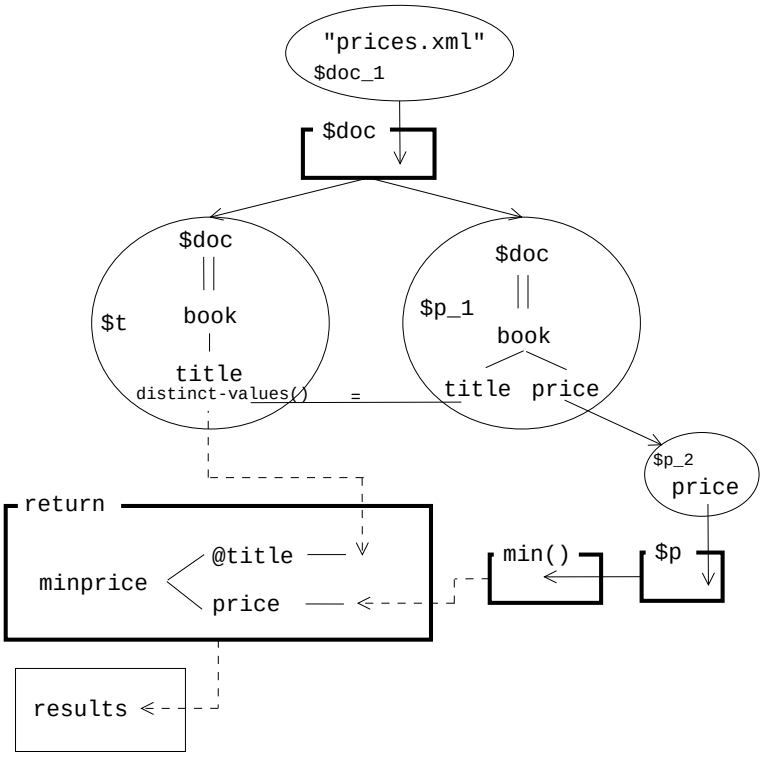


Fig. F.10 – TGV for query XMP 10

Query 11 : *For each book with an author, return the book with its title and authors. For each book with an editor, return a reference with the title and the editor's affiliation*

```
<bib>
{
        for $b in doc("http ://bstore1.example.com/bib.xml")//book[author]
        return <book> { $b/title } { $b/author } </book>
}
{
        for $b in doc("http ://bstore1.example.com/bib.xml")//book[editor]
        return <reference> { $b/title } { $b/editor/affiliation } </reference>
}
</bib>
```

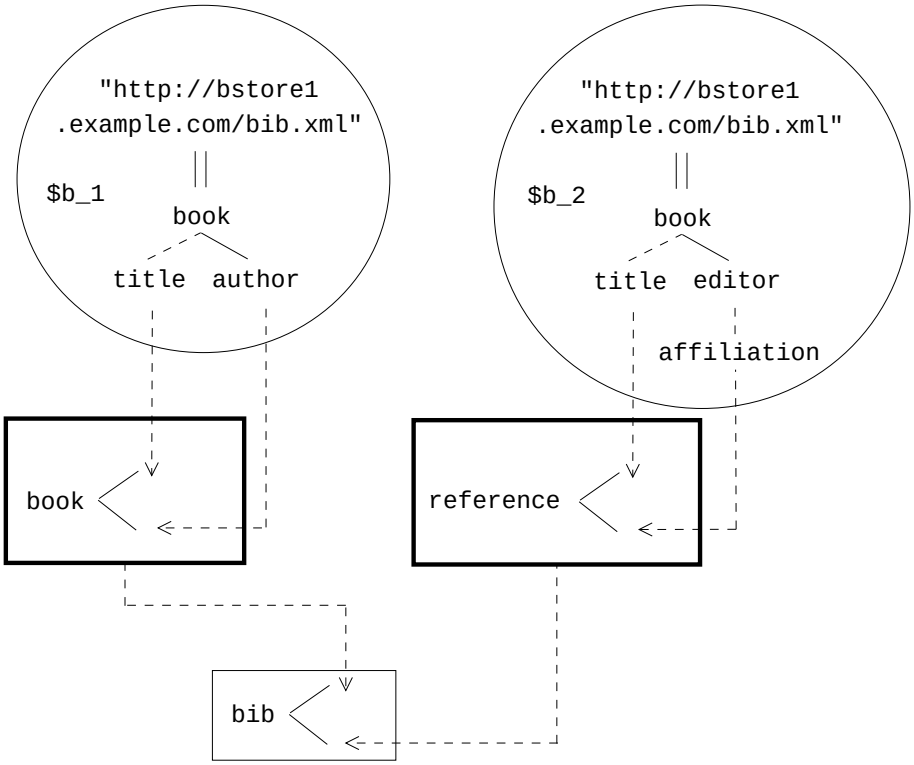


Fig. F.11 – TGV for query XMP 11

Query 12 : *Find pairs of books that have different titles but the same set of authors (possibly in a different order)*

```
<bib>
{
        for $book1 in doc("http ://bstore1.example.com/bib.xml")//book,
                $book2 in doc("http ://bstore1.example.com/bib.xml")//book
        let $aut1 := for $a in $book1/author order by $a/last, $a/first
                return $a
        let $aut2 := for $a in $book2/author order by $a/last, $a/first
                return $a
        where $book1 << $book2
                and not($book1/title = $book2/title)
                and deep-equal($aut1, $aut2)
        return
                <book-pair> { $book1/title } { $book2/title } </book-pair>
}
</bib>
```

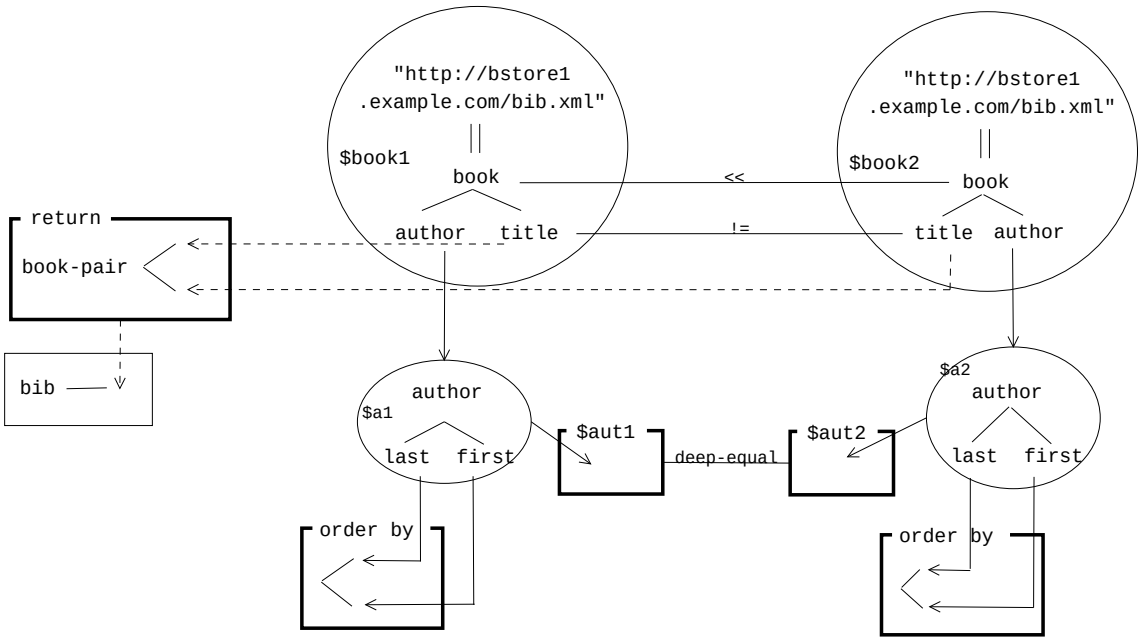


Fig. F.12 – TGV for query XMP 12

F.2 Use Cases TREE

Query 6 : *Make a nested list of the section elements in Book1, preserving their original attributes and hierarchy. Inside each section element, include the title of the section and an element that includes the number of figures immediately contained in the section.*

```
declare function local :section-summary($book-or-section as element()) as element()*
{
        for $section in $book-or-section
        return
                <section>
                        { $section/@* }
                        { $section/title }
                        <figcount>{ count($section/figure) }</figcount>
                        { local :section-summary($section/section) }
                </section>
};
<toc>
        {for $s in doc("book.xml")/book/section
        return local :section-summary($s)}
</toc>
```

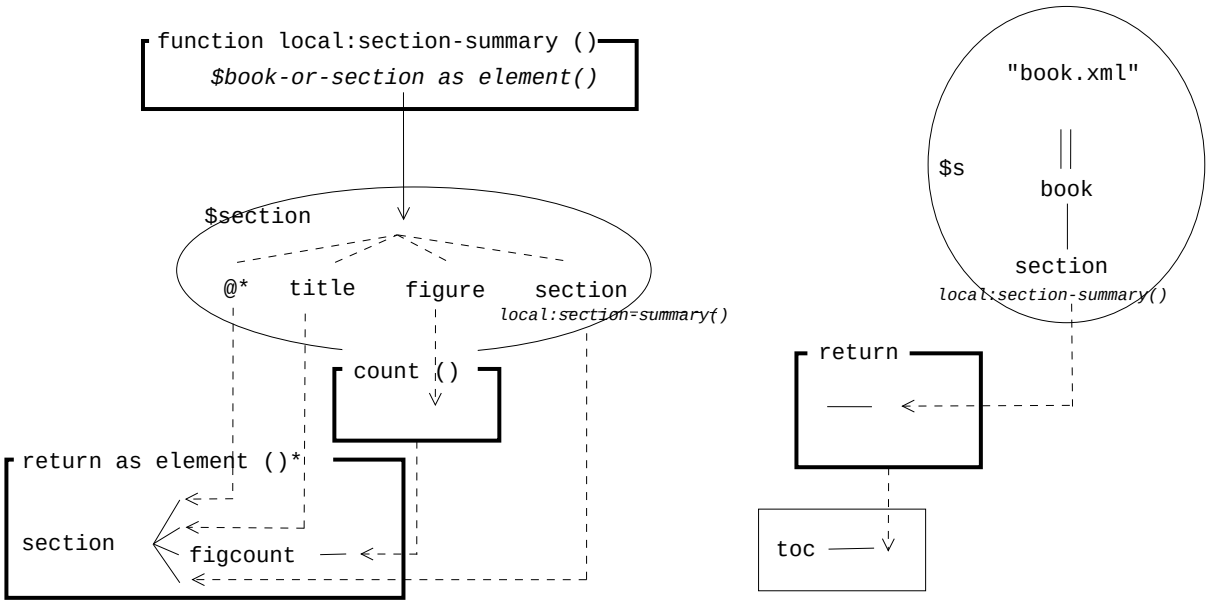


Fig. F.13 – TGV for query TREE 6

F.3 Use Cases SEQ

Query 4 : *In Report1, what happened between the first Incision and the second Incision ?*

```
declare function local :precedes($a as node(), $b as node()) as xs :boolean
        { $a << $b and empty($a//node() intersect $b) } ;
declare function local :follows($a as node(), $b as node()) as xs :boolean
        { $a >> $b and empty($b//node() intersect $a) } ;
<critical_sequence>
{
        let $proc := doc("report1.xml")//section[section.title="Procedure"][1]
        for $n in $proc//node()
        where local :follows($n, ($proc//incision)[1])
                and local :precedes($n, ($proc//incision)[2])
        return $n
}
</critical_sequence>
```

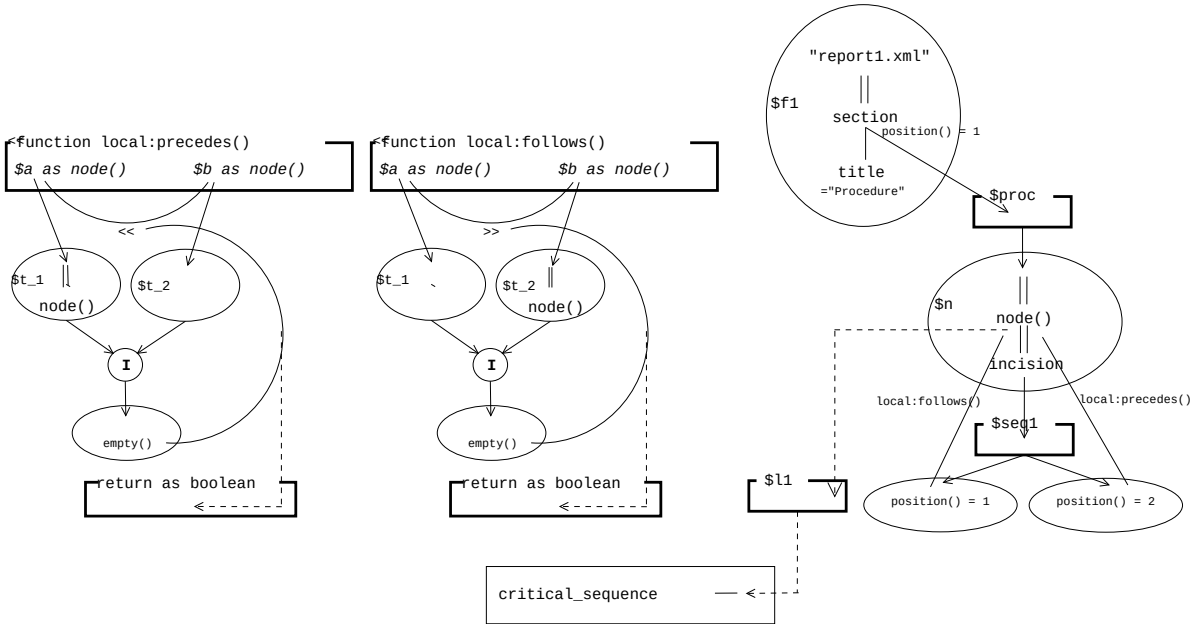


Fig. F.14 – TGV for query TREE 4

F.4 Use Cases R

Query 14 : *List item numbers and average bids for items that have received three or more bids, in descending order by average bid.*

```
<result>
{
        for $i in distinct-values(doc("bids.xml")//itemno)
        let $b := doc("bids.xml")//bid_tuple[itemno = $i]
        let $avgbid := avg($b/bid)
        where count($b) >= 3
        order by $avgbid descending
        return
                <popular_item>
                        <itemno>{ $i }</itemno>
                        <avgbid>{ $avgbid }</avgbid>
                </popular_item>
}
</result>
```

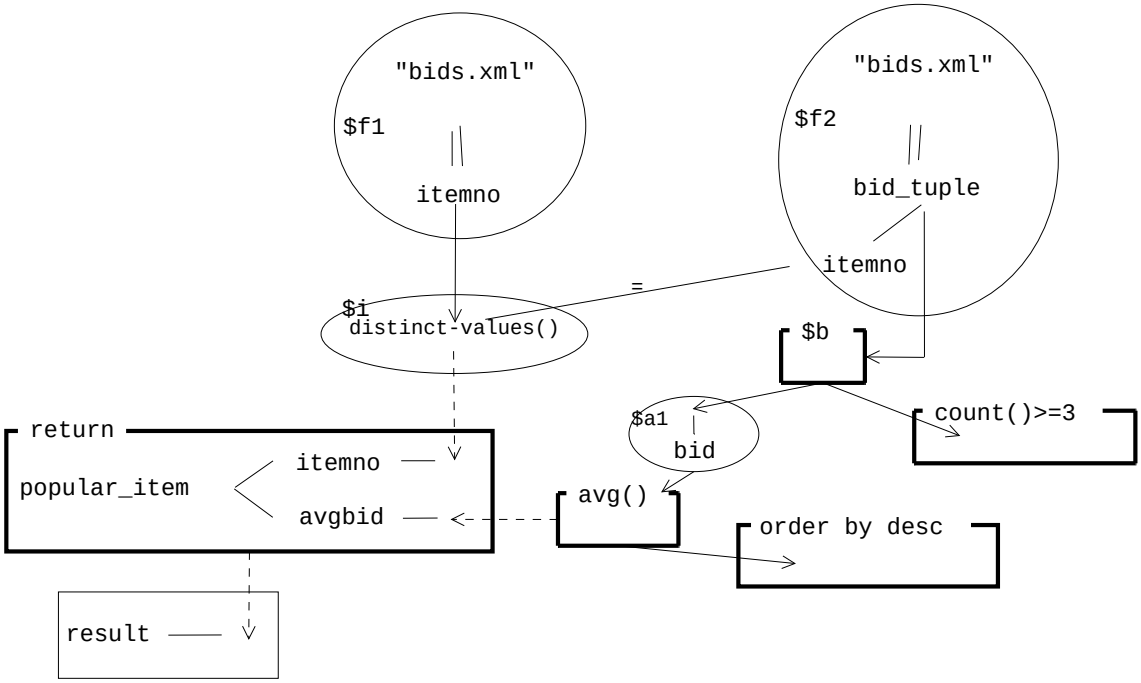


Fig. F.15 – TGV for query R 14

F.5 Use Cases SGML

Query 8 : *Locate all sections with a title that has "is SGML" in it. The string may occur anywhere in the descendants of the title element, and markup boundaries are ignored.*

```
<result>
{
        doc("sgml.xml")//section[.//title[contains(., "is SGML")]]
}
</result>
```

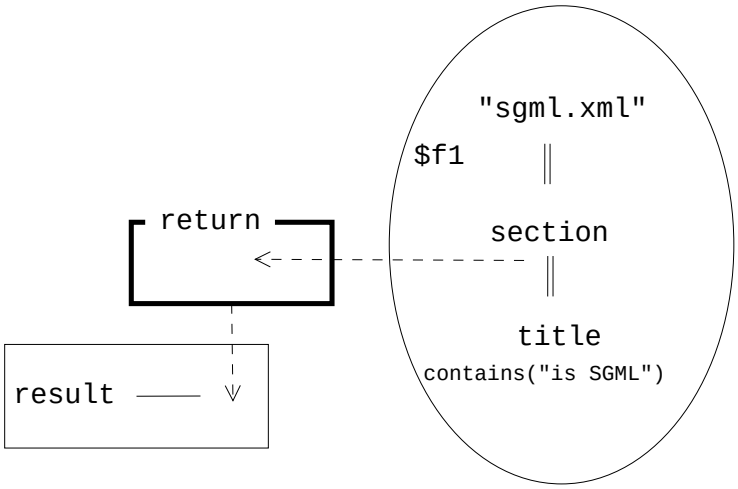


Fig. F.16 – TGV for query SGML 8

F.6 Use Cases STRING

Query 2 : *Find news items where the Foobar Corporation and one or more of its partners are mentioned in the same paragraph and/or title. List each news item by its title and date.*

```
declare function local :partners($company as xs :string) as element()*
        {let $c := doc("company-data.xml")//company[name = $company]
        return $c//partner } ;
let $foobar_partners := local :partners("Foobar Corporation")
for $item in doc("string.xml")//news_item
where
    some $t in $item//title satisfies
        (contains($t/text(), "Foobar Corporation")
        and (some $partner in $foobar_partners satisfies
            contains($t/text(), $partner/text())))
    or (some $par in $item//par satisfies
        (contains(string($par), "Foobar Corporation")
        and (some $partner in $foobar_partners satisfies
            contains(string($par), $partner/text()))))
return
        <news_item> { $item/title } { $item/date } </news_item>
```

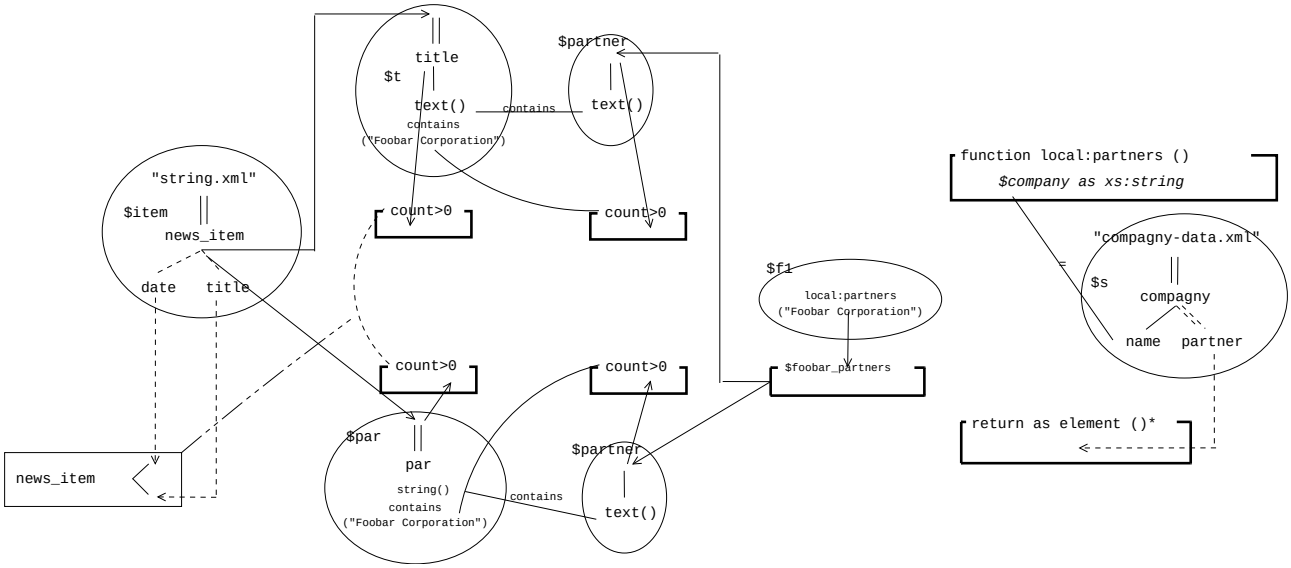


Fig. F.17 – TGV for query STRING 2

F.7 Use Cases NS

Query 7 : *Select the homepage of all auctions where both seller and high bidder are registered at the same auctioneer.*

```
declare namespace ma = "http ://www.example.com/AuctionWatch";
<Q7 xmlns :xlink="http ://www.w3.org/1999/xlink">
{
        for $a in doc("auction.xml")//ma :Auction
         let $seller_id := $a/ma :Trading_Partners/ma :Seller/* :ID,
                $buyer_id := $a/ma :Trading_Partners/ma :High_Bidder/* :ID
         where namespace-uri($seller_id) = namespace-uri($buyer_id)
         return
                $a/ma :AuctionHomepage
}
</Q7>
```

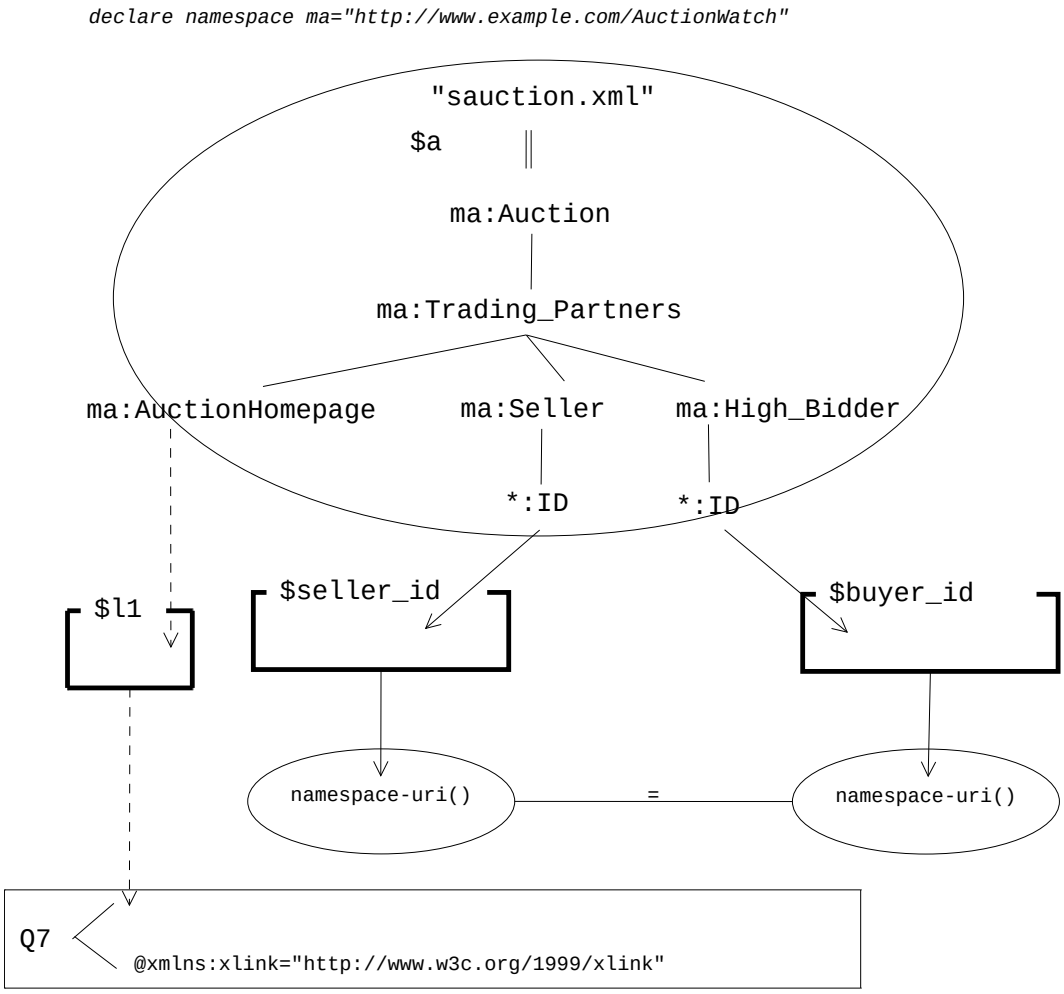


Fig. F.18 – TGV for query NS 7

F.8 Use Cases PARTS

Query 1 : *Convert the sample document from "partlist" format to "parttree" format (see DTD section for definitions). In the result document, part containment is represented by containment of one <part> element inside another. Each part that is not part of any other part should appear as a separate top-level element in the output document.*

```
declare function local :one_level($p as element()) as element()
{
        <part partid="{ $p/@partid }" name="{ $p/@name }" >
        {
                for $s in doc("partlist.xml")//part
                where $s/@partof = $p/@partid
                return local :one_level($s)
        }
        </part>
};

<parttree>
{
        for $p in doc("partlist.xml")//part[empty(@partof)]
        return local :one_level($p)
}
</parttree>
```

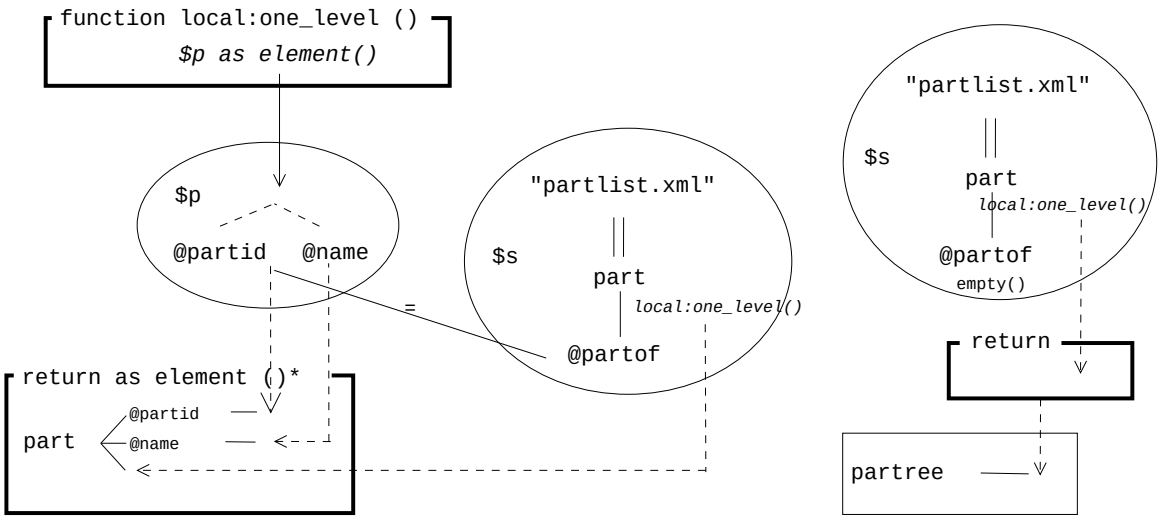


Fig. F.19 – TGV for query PARTS 1

Annexe G

Propositions de modélisation

Nous proposons ici quelques représentations de requêtes de typage et de mise à jour. Ces représentations sont des perspectives de recherche, elles sont présentes à titre d'exemple.

G.1 Use Case STRONG

Query 4 : *Determine whether the postal code or zip code for a purchase order is right.*

```
declare function local :address-ok($a as element(*, ipo :Address)) as xs :boolean {
        typeswitch ($a)
        case $zip as element(*, ipo :USAddress) return zok :zip-ok($zip)
        case $postal as element(*, ipo :UKAddress) return pok :postal-ok($postal)
        default return false() } ;
let $shipTo := doc("ipo.xml")/element(ipo :purchaseOrder)/shipTo
let $billTo := doc("ipo.xml")/element(ipo :purchaseOrder)/billTo
return local :address-ok($shipTo) and local :address-ok($billTo)
```

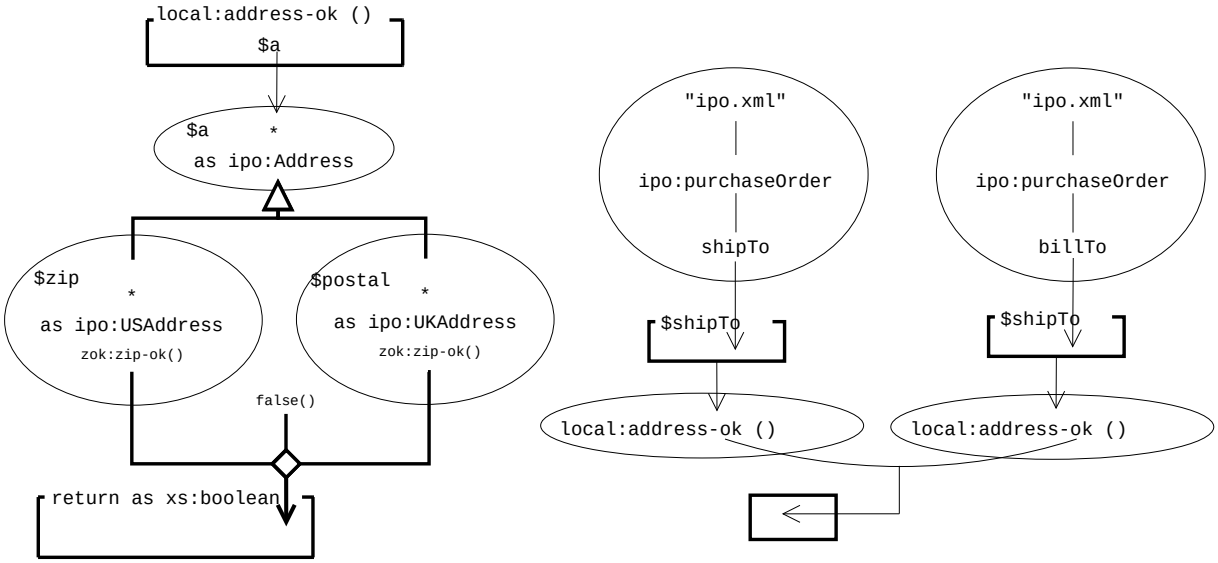


Fig. G.1 – TGV for query STRONG 4

Nous pouvons observer sur la figure G.1 la définition d'une fonction à gauche, et un double appel à droite à l'aide des résultats présents dans les motifs d'arbre d'agrégation «$shipTo» et «$billTo». Si les deux résultats de fonctions sont vrais, alors *vrai* est retourné, sinon *faux*. Les noms d'élement typés («*element(ipo :...)*») sont canonisés.
Dans la définition de fonction, nous pouvons observer un nouveau type d'hyperlien correspondant à aux «*typeswitch*». Le triangle exprime le typage de l'élément d'entrée, le nœud cible contenu dans les motifs d'arbre donne le typage de test, si la contrainte de typage est vérifiée alors la fonction «*zip-ok()*» est appliquée (avec le namespace correspondant), le résultat choisi est symbolisé par l'hyperlien avec le losange. Le résultat de la fonction est de type booléen.

G.2 Use Case UPDATE

Query 1 : *Add a new user (with no rating) to the users.xml view.*

```
insert {<user_tuple>
            <userid>U07</userid>
            <name>Annabel Lee</name>
        </user_tuple>}
into doc("users.xml")/users
```

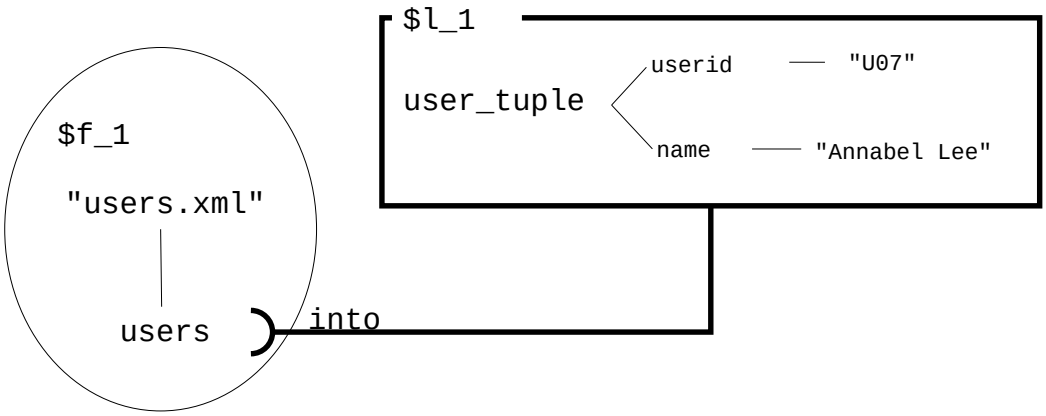


Fig. G.2 – TGV for query UDPATE 1

La figure G.2 montre un exemple de représentation de mise à jour. Cette insertion de données dans le motif d'arbre $l_1 est symbolisé par l'hyperlien de mise à jour avec comme fin de ligne un «*réceptacle*» (représentation du format UML). Il permet ainsi de représenter une insertion du motif d'arbre d'agrégation dans le motif d'arbre source. L'annotation «*into*» permet de représenter le type d'insertion (il existe INTO/AFTER/BEFORE). Nous pouvons annoter la position (first, last).
Une suppression «*DELETE*» serait symbolisée par une croix et une modification «*REPLACE*» par un rond.

”Beaucoup encore il te reste à apprendre.”
Yoda (Épisode II)

”Que la recherche soit avec toi, à jamais.”
Obi-wan Kenobi (Épisode IV)

”Le rapport est fini p’tit gars ! Fais moi péter cette soutenance et on va au pot !”
Han Solo (Épisode IV)

”La recherche est avec toi jeune doctorant, mais tu n’es pas encore un chercheur.”
Dark Vador (Épisode V)

”Fais ta thèse, ou ne la fait, il n’y a pas d’essai.”
Yoda (Épisode V)

”Difficile à voir. Toujours en mouvement est la recherche.”
Yoda (Épisode V)

Optimisation Extensible dans un Médiateur de Données Semi-Structurées

Résumé : *Cette thèse propose un cadre d'évaluation pour des requêtes XQuery dans un contexte de médiation de données XML. Un médiateur doit fédérer des sources de données distribuées et hétérogènes. A cette fin, un modèle de représentation des requêtes est nécessaire. Ce modèle doit intégrer les problèmes de médiation et permettre de définir un cadre d'optimisation pour améliorer les performances. Le modèle des motifs d'arbre est souvent utilisé pour représenter les requêtes XQuery, mais il ne reconnaît pas toutes les spécifications du langage. La complexité du langage XQuery fait qu'aucun modèle de représentation complet n'a été proposé pour reconnaître toutes les spécifications. Ainsi, nous proposons un nouveau modèle de représentation pour toutes les requêtes XQuery non typées que nous appelons TGV. Avant de modéliser une requête, une étape de canonisation permet de produire une forme canonique pour ces requêtes, facilitant l'étape de traduction vers le modèle TGV. Ce modèle prend en compte le contexte de médiation et facilite l'étape d'optimisation. Les TGV définis sous forme de Types Abstraits de Données facilitent l'intégration du modèle dans tout système en fonction du modèle de données. De plus, une algèbre d'évaluation est définie pour les TGV. Grâce à l'intégration d'annotations et d'un cadre pour règles de transformation, un optimiseur extensible manipule les TGV. Celui-ci repose sur des règles transformations, un modèle de coût générique et une stratégie de recherche. Les TGV et l'optimiseur extensible sont intégrés dans le médiateur XLive, développé au laboratoire PRiSM.*
Mots clés : *XML, XQuery, Motifs d'arbre, TGV, Optimisation extensible*

Extensible Optimization in an XML Mediator

Abstract : *This thesis proposes to evaluate XQuery queries into a mediation context. This mediator must federate several heterogeneous data sources with an appropriate query model. On this model, an optimization framework must be defined to increase performance. The well-known Tree Pattern model can represent a subset of XPath queries in a tree form. Because of the complexity of XQuery, no model has been proposed that is able to represent all the structural components of the language. Then, we propose a new logical model for XQuery queries called TGV. It aims at supporting the whole XQuery specification for un-typed queries. Before modelling, a canonization step transforms XQuery queries into a canonical form in order to check more XQuery specifications. This form allows us to translate in a unique way queries into our TGV model. This model takes into account a distributed heterogeneous context and eases the optimization process. It integrates transformation rules, cost evaluation, and therefore, execution of XQuery queries. The TGV can be used as a basis for processing XQuery queries, since it is flexible, it provides abstracts data types which can be implemented according to the underneath data model. Moreover, it allows user-defined annotating and also cost-related annotating for cost estimation. Although the model will be useful, it relies on XQuery complicated specifications. TGV are illustrated in this thesis with several figures on W3C's use-cases. Finally, a framework to define transformation rules is added to the extensible optimizer to increase the XLive mediator performances. The XLive mediation system has been developed at the PRiSM Laboratory.*
Keywords : *XML, XQuery, Tree Patterns, TGV, Extensible Optimization*