Langages de requêtes pour XML à base de patterns : conception, optimisation et implantation

Cédric Miachon

▶ **To cite this version:**

Cédric Miachon. Langages de requêtes pour XML à base de patterns : conception, optimisation et implantation. Informatique [cs]. Université Paris Sud - Paris XI, 2006. Français. NNT : . tel-00603376

HAL Id: tel-00603376
https://theses.hal.science/tel-00603376

Submitted on 24 Jun 2011

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.

N° d'ordre: 8544

THÈSE

présentée

devant l'Université Paris-Sud 11

pour obtenir

le grade de : Docteur de l'Université Paris-Sud 11
Mention Informatique

par

Cédric Miachon

Équipe d'accueil : Bases de Données - LRI
École Doctorale d'Informatique de Paris-Sud 11
Titre de la thèse :

Langages de requêtes pour XML à base de patterns : conception, optimisation et implantation

Soutenue le 13 décembre 2006 devant la commission d'examen composée de :

Mme	Zohra Bellahsene	Examinatrice
Mme	Véronique Benzaken	Directrice de thèse
M.	Giuseppe Castagna	Directeur de thèse
Mme	Christine Paulin	Présidente
M.	Jacques Le Maitre	Rapporteur
Mme	Sophie Tison	Rapportrice

À Colette.

Remerciements

Mes remerciements s'adressent à Véronique Benzaken et à Giuseppe Castagna qui ont bien voulu encadrer ma thèse et mon stage de DEA. Leurs conseils, leur patience et leurs questions m'auront été très utiles. J'ai pu, grâce à eux, mener à bien cette thèse, et prendre goût à tous les aspects de la recherche, ainsi qu'à l'enseignement.

Sophie Tison et Jacques Le Maitre m'ont fait l'honneur d'être rapporteurs de cette thèse. Je leur en suis très reconnaissant. Leurs remarques et questions m'ont été très utiles. Je tiens également à remercier Zohra Bellahsene et Christine Paulin d'avoir accepté de participer à mon jury.

Je tiens à remercier les membres de l'équipe Langages du LIENS et de l'équipe Bases de Données du LRI, pour leur sympathie, leurs conseils, leurs questions et surtout leurs réponses, et pour tous les échanges qui ont eu lieu ces trois dernières années. Notamment Alain Frisch, Dario Colazzo, Nicole Bidoit, Kim Nguyễn, Marwan Burelle et Matthieu Objois . Je remercie pour les mêmes raisons également Ioana Manolescu, Bertrand, Sammy, Ignacy, Nicolas, ...

Et pour finir je remercie aussi Amélie, ma famille, et plus particulièrement ma grand-mère qui n'a pas pu voir l'aboutissement de ce travail.

Table des matières

1 Introduction 11

2 État de l'art 17

2.1 XML : Un format de documents 18

2.1.1 Présentation 18

2.1.2 Validation par typage de documents XML 20

2.1.3 XHTML : un format de pages web 23

2.2 Langages de manipulation de documents XML 24

2.2.1 Déconstructeurs et itérateurs 25

2.2.2 Lorel et SgmlQL 30

2.2.3 XML-QL 31

2.2.4 XQuery 32

2.2.5 Cω et XLinq 34

2.2.6 XDuce 34

2.3 $\mathbb{C}$Duce 36

2.3.1 Algèbre de types 37

2.3.2 Filtrage par motifs 41

2.4 Conclusion 44

3 Le langage de requêtes $\mathbb{C}$QL 47

3.1 Motivations 47

3.2 Description du langage de requêtes 48

3.2.1 Syntaxe 48

3.2.2 Sémantique 53

3.2.3 Exemples de requêtes 57

3.3 Conclusion 64

4 Optimisations 65

4.1 Motivations 65

4.2 Traduction et désimbrication des projections en filtrage par motifs 66

4.2.1 Algorithme de traduction 66
4.2.2 Préservation de la sémantique 70
4.2.3 Préservation du typage 79
4.2.4 Exemple 82
4.3 Autres optimisations 83
4.3.1 Transformation en motifs d'une partie de la condition . . . 84
4.3.2 Consolidation des motifs 85
4.3.3 Remontée des sélections 86
4.4 Conclusion 87

5 Performances **89**
5.1 Motivations 89
5.2 Approche *Jeux de tests* 90
5.2.1 Sur les *XML Query Use Cases* 91
5.2.2 Sur XMark 103
5.3 Approche *micro-benchmarks* 107
5.3.1 Présentation des requêtes 110
5.3.2 Résultats de ℂQL 115
5.3.3 Quelques performances comparées 119
5.4 Conclusion 121

6 Interface graphique : Pattern by Example **123**
6.1 Motivations 123
6.2 Différents langages de requêtes graphiques 124
6.2.1 QBE : un langage graphique pour le relationnel 125
6.2.2 Langages graphiques pour XML 126
6.3 Présentation de PBE 129
6.3.1 Visite guidée 129
6.4 Description formelle de PBE 138
6.4.1 Définitions 139
6.4.2 Syntaxe des tableaux 141
6.4.3 Sémantique 143
6.4.4 Algorithme de traduction d'une requête PBE en une requête ℂQL 144
6.5 Conclusion 158

7 Conclusion et perspectives **161**
7.1 Conclusion 161

7.2 Perspectives 162

Liste des figures **164**

Bibliographie **169**

Chapitre 1

Introduction

L'objectif initial du travail de recherche présenté dans cette thèse est la création d'un langage de requêtes adapté pour l'interrogation de données semi-structurées, utilisant le mécanisme de filtrage par motifs avec typage fort et statique. Nous considérerons par la suite que des données au format XML et des données semi-structurées sont synonymes. Ce travail a conduit à la définition du langage de requêtes $\mathbb{C}$QL, à son implantation dans le langage $\mathbb{C}$Duce, à l'ajout d'optimisations validées par une série de mesures opérées à partir de benchmarks et à la définition et l'implantation d'une interface graphique, PBE, adaptée à l'interrogation simplifiée de documents XML.

L'explosion récente de l'Internet conduit à un échange croissant d'informations. Ces informations sont traitées puis restructurées à partir de sources de données multiples et hétérogènes [ABS00].

XML (« *Extensible Markup Language* ») [W3C00] est un standard proposé par le W3C (« *World Wide Web Consortium* »). XML permet de faciliter l'échange et le partage de textes et d'informations structurées. Par exemple en permettant de séparer le contenu (les données) du contenant (la présentation des données). XML offre une meilleure compréhension des informations pertinentes relatives à un document. Il constitue, en outre, une simplification de SGML [ISO86]. XML est devenu le standard pour l'échange et la manipulation de ces données sur le Web, succès sûrement dû à son caractère facilement lisible, et à sa nature autodescriptive qui permet de mélanger à la fois le contenu du document et sa propre structure.

Un document XML est constitué d'éléments, d'attributs et de texte. On peut le typer via une DTD, ou XML Schema. Dans l'exemple de la figure 6.1, `manuscrit`, `titre`, `date`, `auteur`, `prenom`, `nom` et `mel` sont des étiquettes, tandis que *type* est un attribut, et le texte est représenté ici en caractères gras. Les ba-

```
<manuscrit type=" these">
    <titre> Langages de requêtes pour XML ...</titre>
    <date> 13 décembre 2006</date>
    <auteur>
        <prenom> Cédric</prenom>
        <nom> Miachon</nom>
        <mel> miachon@lri.fr</mel>
    </auteur>
</manuscrit>
```

FIG. 1.1 – Exemple de document XML

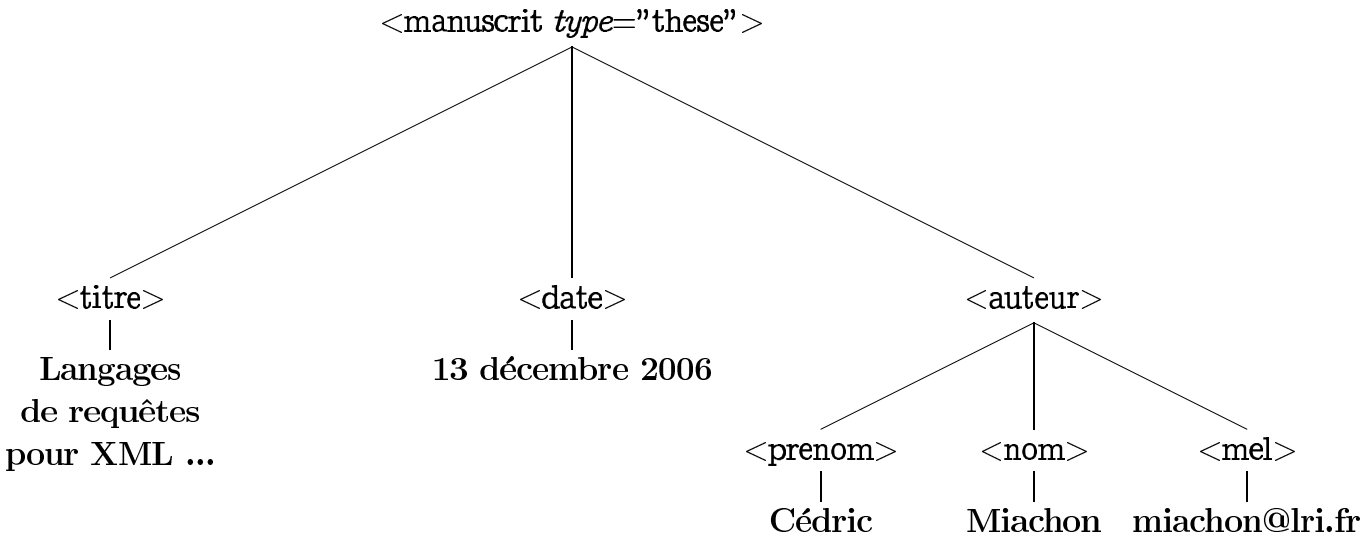


FIG. 1.2 – Représentation sous forme d'arbre d'un document XML

lises $< \cdots >$ et $< / \cdots >$ indiquent respectivement le début et la fin d'un élément ; et contiennent une étiquette et éventuellement des attributs. Un élément peut contenir un mélange de texte ou d'autres éléments. Une des conditions de bonne formation des documents XML est celle du bon parenthésage de ces balises, ce qui permet de voir les documents comme des arbres. Dans cet exemple, la racine de l'arbre de la figure 1.2 est l'élément d'étiquette `<manuscrit>`.

Langages de manipulation

Différents langages ont été conçus pour permettre d'extraire de l'information à partir de ces documents, ou pour manipuler des données, de manière, par exemple à n'en garder qu'une partie, ou alors à les combiner avec d'autres données extraites d'autres documents.

Il est possible de classer ces langages selon différentes catégories :

- les langages de programmation comme XSLT [Cla99] qui est non typé et qui au moyen de règles permet de transformer un document XML en un autre.

Un programme écrit en XSLT/XPath a pour particularité d'être lui-même un document XML.

- les langages de programmation fonctionnels comme XDuce [HP00], ℂDuce [BCF03], et les langages objet comme XTatic [GLPS05], qui, grâce à un typage statique, précis et fort, sont adaptés à la manipulation sûre et efficace de documents XML.
- les langages de requêtes, parmi lesquels on compte XQuery [BCF+01], qui est le langage de requêtes standard proposé par le W3C.

Chacun de ces langages offrent entre autre deux types de primitives [Cas05] : les déconstructeurs et les itérateurs que nous présentons.

Déconstructeurs

Les déconstructeurs servent à extraire de l'information en capturant des sous-parties de documents XML. Une primitive de déconstruction est illustrée par les expressions de chemin du langage XPath.

XPath [CD99] est un langage qui permet de naviguer à l'intérieur d'un arbre, représentant le document XML, en suivant des "chemins" dans l'arbre. Il est utilisé par XSLT et XQuery. Un exemple de chemins XPath est `manuscrit/auteur/mel` qui appliqué à l'arbre XML de la figure 1.2, renverra le sous-arbre `<mel>`**miachon@lri.fr**`</mel>`. Il existe aussi comme déconstructeurs la navigation de Lorel [AQM+97] qui est très similaire à XPath.

Une autre classe de déconstructeurs est le filtrage par motifs (commun aux langages fonctionnels) qui va appliquer sur un arbre un certain motif contenant des variables, et va les lier à des sous-arbres selon certains critères. Par exemple le motif : `<auteur>` [`<prenom>`[p] `<nom>`[n] m] appliqué au sous-arbre de balise racine `<auteur>` va capturer pour p le texte **Cédric**, pour n **Miachon**, et pour m le sous-arbre $\langle$`mel`$\rangle$ [**miachon@lri.fr**]. Ce filtrage par motifs est utilisé dans des langages comme XDuce et ℂDuce.

L'intérêt du filtrage par motifs est de pouvoir capturer en une seule expression différentes sous-parties (ou sous-arbres) d'un document XML. Avec des déconstructeurs à la XPath, pour capturer autant de sous-arbres différents, il faudra autant d'expressions de chemins.

Itérateurs

Chacun de ces langages dispose au moins d'un itérateur qui permet d'appliquer un déconstructeur sur des collections d'arbres. En XSLT, il s'agit des règles

qui s'appliquent à une expression XPath ; tout comme l'opérateur `for let where return` (qui est appelé FLWR et se prononce « *flower* ») de XQuery. Dans les langages fonctionnels on trouve le `filter` de XDuce, le `transform` et le `xtransform` de ℂDuce. Tous ces opérateurs sont syntaxiquement et sémantiquement différents, mais se ressemblent fortement car ils itèrent un déconstructeur sur une collection d'éléments dans le but d'en capturer une partie. Nous insisterons dans ce manuscrit sur le langage ℂDuce car c'est dans ce cadre que la thèse s'est déroulée.

ℂDuce est un langage de programmation pour XML fonctionnel, fortement et statiquement typé. Il respecte les standards usuels pour XML (Namespaces, Unicode, XML Schema, DTD,...), et est distribué pour différents systèmes d'exploitation à l'adresse `www.cduce.org`.

Les deux types de déconstructeurs sont utiles et complémentaires [Cas05]. Il n'existe pourtant qu'un seul langage de requêtes sur des documents XML qui offre à la fois un itérateur qui utilise ces deux types de déconstructeur. Il s'agit de XML-QL [DFF+98, DFF+99], mais n'est cependant pas un langage typé, et qui n'a pas la même expressivité que les algèbres de motifs et de types de ℂDuce. L'objectif majeur de cette thèse est de concevoir ainsi un langage de requêtes fortement typé qui sera basé sur des expressions de chemins et sur le filtrage par motifs, très expressifs, de ℂDuce. Outre la conception d'un tel langage, nous étudierons quelques optimisations. Nous les vérifierons par des tests, et finalement, nous définirons une interface graphique.

Plan détaillé

La première partie du travail de cette thèse consiste en la définition d'un langage de requêtes pour XML qui est appelé ℂQL. Ce langage utilise contrairement à d'autres langages de requêtes ou de manipulation de documents XML, un itérateur utilisant à la fois le filtrage par motifs et le déconstructeur de chemins à la XPath.

Le chapitre 2 propose un état de l'art du domaine des langages de requêtes pour le XML. Plus précisément, nous présenterons le format XML, avec les méthodes de validation qui existent. Nous présenterons différents langages en insistant sur leurs déconstructeurs et leurs itérateurs. Nous détaillerons longuement à la fin de ce chapitre le langage ℂDuce qui sert de langage de base pour la définition de ℂQL, en lui apportant ainsi son algèbre de types et l'expressivité de son filtrage par motifs.

Dans le chapitre 3 nous présenterons le langage ℂQL avec sa syntaxe et sa sémantique. Ce langage permet d'écrire à la fois des requêtes utilisant le

filtrage par motifs et des expressions de chemins par navigation descendante. Nous montrerons son expressivité à travers quelques exemples tiré des publications [BCM04, CDu05].

Le chapitre 4 est consacré aux optimisations apportées à ℂQL. Nous étudierons une réécriture de projections en motifs, et nous prouverons que la sémantique et le typage de cette réécriture sont préservés. Nous apporterons une transformation des conditions en motifs, et une optimisation inspirée de la descente des sélections inspirée du modèle relationnel [BCM05].

Dans le chapitre 5 nous étudierons les performances en temps de ℂQL par rapport à d'autres implantations en XQUERY, en suivant des approches différentes : la première sur des jeux de tests et la seconde sur des « microbenchmarks ». Et nous évaluerons l'impact des différentes optimisations sur ℂQL, et nous montrerons que les requêtes utilisant le filtrage par motifs sont plus rapides que celles n'utilisant que les projections [BCM05, MMM06].

La deuxième partie de ce travail est présentée dans le dernier chapitre 6. Il s'agit de la création d'une interface graphique, appellée PBE, à partir de tableaux étendus de QBE [Zlo75, Zlo77]. Dans un souci d'abstraction par rapport au langage de requêtes, nous proposons un langage graphique qui rendra transparente l'écriture de motifs à un utilisateur. Et nous nous intéresserons aux manières de générer des requêtes en ℂQL, à partir des tableaux graphiques PBE.

Bibliographie

[BCM04] Véronique Benzaken, Giuseppe Castagna, and Cédric Miachon. CQL : a pattern-based query language for XML. In *Actes des 20es Journées des Bases de Données Avancées (BDA)*, 2004.

[BCM05] Véronique Benzaken, Giuseppe Castagna, and Cédric Miachon. A full pattern-based paradigm for XML query processing. In *PADL 05, 7th International Symposium on Practical Aspects of Declarative Languages*, number 3350 in Lecture Notes in Computer Science, pages 235–252. Springer, January 2005.

[CDu05] The CDucers. ℂDuce : an XML-centric programming language. In *Actes des 21es Journées des Bases de Données Avancées (BDA)*, 2005.

[MMM06] Ioana Manolescu, Cédric Miachon, and Philippe Michiels. Towards micro-benchmarking XQuery. In *First International Work-*

shop on Performance and Evaluation of Data Management Systems (EXPDB 2006), Chicago, 2006. Collocated with ACM SIGMOD/PODS.

Chapitre 2

État de l'art

Torniamo all'antico e sarà un progresso. (Verdi)

Sommaire

2.1 XML : Un format de documents 18
2.1.1 Présentation 18
2.1.2 Validation par typage de documents XML 20
2.1.3 XHTML : un format de pages web 23
2.2 Langages de manipulation de documents XML 24
2.2.1 Déconstructeurs et itérateurs 25
2.2.2 Lorel et SgmlQL 30
2.2.3 XML-QL 31
2.2.4 XQuery 32
2.2.5 Cω et XLinq 34
2.2.6 XDuce 34
2.3 $\mathbb{C}$Duce 36
2.3.1 Algèbre de types 37
2.3.2 Filtrage par motifs 41
2.4 Conclusion 44

Nous présenterons dans ce chapitre XML et les différentes techniques de validation qui lui sont associées.

Nous ferons ensuite un état de l'art des langages de manipulation en présentant les différents déconstructeurs et itérateurs qu'offrent ces langages. Puis nous ferons une présentation de $\mathbb{C}$Duce en détaillant le filtrage par motifs et ses algèbres de types et de motifs.

2.1 XML : Un format de documents

XML (« *Extensible Markup Language* ») est un format de représentation de documents. XML est une forme simplifiée de SGML (« *Standard Generalized Markup Language* »), qui a été officiellement défini en 1986 [ISO86] mais qui est utilisé depuis les années 1960. Plusieurs formats dérivent de SGML comme HTML et DocBook, et qui ont été redéfinis à travers XML. XML a été proposé en 1998 comme standard par le W3C (« *World Wide Web Consortium* ») [W3C00], et s'est rapidement imposé grâce notamment à sa simplicité. Bien que de nombreuses entreprises l'aient déjà adopté, les aspects liés au typage des documents sont peu exploités.

2.1.1 Présentation

XML permet de définir des documents qui contiennent à la fois le contenu et la structure d'une information.

Un document XML est constitué d'éléments, d'attributs et de texte. On peut le typer via une DTD ou XML Schema.

L'exemple de la figure 2.1 servira par la suite d'exemple principal. Il est tiré des « *XML Query Use Cases* » [CFF+03], et sert de base pour une série de requêtes (par exemple celle de la figure 2.9 à la page 33). Dans cet exemple, il y a différents éléments comme la balise `<book year="1994">` qui contient une balise `book` et un attribut *year*, ce dernier contenant du texte. Cet élément a des éléments fils comme `<title>`**TCP/IP Illustrated**`</title>`, ou

```
<author>
  <last>Stevens</last>
  <first>W.</first>
</author>
```

qui contient lui-même deux autres éléments fils `<last>` et `<first>`.

Toute balise ouverte doit être fermée, et comme un élément peut contenir d'autres éléments, il faut vérifier que la condition de bon parenthésage est respectée. Si cette condition est respectée on dira du document XML qu'il est *bien formé*, et sinon *mal formé*. Voici un exemple de document mal formé :

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
  </book>
    </title>
```

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last>
      <first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>

  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
    <author>
      <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>

  <book year="2000">
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
    <author>
      <last>Buneman</last>
      <first>Peter</first>
    </author>
    <author>
      <last>Suciu</last>
      <first>Dan</first>
    </author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>

  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <editor>
      <last>Gerbarg</last>
      <first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

FIG. 2.1 – Document XML de référence représentant une bibliographie

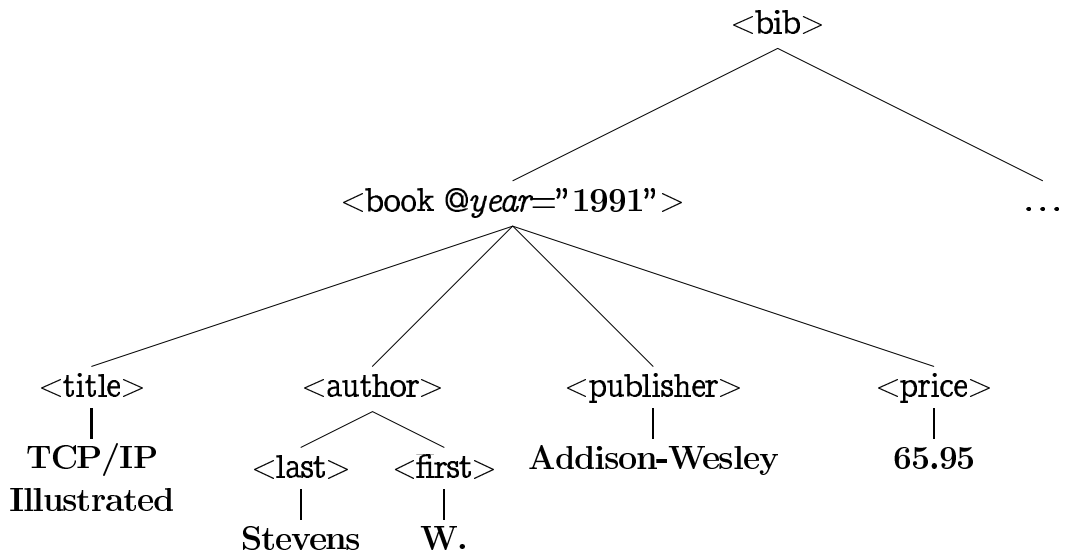


FIG. 2.2 – Représentation sous forme d'arbre du document XML de référence

L'élément `book` contient la balise `title`, mais `title` est fermée après la balise `book`, ce qui est incorrect. Il manque aussi la fermeture de la balise `bib`.

Cette condition de bon parenthèsage permet de représenter les documents XML comme des arbres. L'arbre de la figure 2.2 représente ainsi notre document de référence.

2.1.2 Validation par typage de documents XML

En pratique la contrainte imposée pour XML de bon parenthésage ne suffit pas, et il est fréquent d'associer un type à un document XML dans le but de vérifier que ces documents ont bien le schéma attendu.

Ces documents peuvent donc être typés au moyen d'une DTD ou de schémas. Cependant la plupart des langages largement utilisés pour manipuler de tels documents n'exploitent pas le typage du document. Les langages tels XSLT [Cla99] ne sont pas typés. Ceci peut paraître surprenant pour un standard de représentation d'informations sur l'Internet : assurer la manipulation la plus fiable possible lors de traitements de documents XML ou d'extractions de données est un point crucial pour garantir l'intégrité et la qualité des échanges à travers différents services.

Valider un document consiste à vérifier des contraintes sur la structure et le contenu.

Plusieurs langages de validation de documents XML existent, notamment les DTD [W3C00], XML SCHEMA [W3C01b], RELAX-NG [Mur00, rel], DSD [KMS02], et TREX [Cla01]. On trouvera dans [MLM01, LC00] une comparaison de leur expressivité.

Il existe également la possibilité de valider des documents par le biais d'un

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

FIG. 2.3 – DTD associée à un document XML de référence

système de types intégré à un langage de programmation. Les types ont plusieurs interprétations classiques qui correspondent à des points de vue différents. Ici, dans le cadre de la validation de documents XML, on voit les types comme des contraintes sur la structure et le contenu de documents. Un type définit donc un ensemble de documents, de valeurs du langage. Ce point de vue est proche de celui des bases de données, où le schéma d'une base décrit ses états admissibles, contraint les données et permet l'optimisation de requêtes.

Nous illustrerons cette validation de documents XML par des types spécifiques et internes à un langage, dans les parties consacrées plus loin dans ce chapitre à XDUCE et ℂDuce. Nous présentons ci-après les DTD et XML SCHEMA.

2.1.2.1 DTD

Une DTD (« *Document Type Definition* ») [W3C00], est un document permettant de décrire un modèle de documents XML, et la structure qu'un document doit avoir (hiérarchie des champs, paramètres, type des données,...).

Un document XML étant constitué d'éléments, d'attributs et de texte, une DTD permet de poser des contraintes sur des éléments et leur contenu. Ces contraintes sont décrites au moyen de l'entité < !ELEMENT>, qui porte le nom de l'étiquette de la balise racine.

Par exemple, dans les « *XML Query Use Cases* », la DTD de la figure 2.3 est proposée pour le document de référence XML de la figure 2.1.

À la première ligne, on trouve dans cet exemple, l'élément racine décrit comme étant l'entité `bib` et contenant une expression régulière (`(book*)` signifiant que l'élément `bib` ne peut avoir comme fils qu'une suite potentiellement vide (représentée par l'étoile de Kleene) d'entités `book`). On ne peut définir qu'une entité par nom de balise ; ainsi tous les éléments ayant pour étiquette `title` doivent ne contenir qu'une chaîne de caractères. Nous ne pouvons pas dans le cadre d'une

DTD avoir deux entités exprimant deux expressions régulières différentes portant sur la même balise ; ce qui en limite le pouvoir expressif.

La deuxième ligne décrit l'élément `book` comme ayant pour fils une suite ordonnée composée d'un élément `title`, puis, soit d'une suite non vide (représentée par le signe +) d'éléments `author`, soit d'une suite non vide d'éléments `editor`, et ensuite d'un élément `publisher` et finalement un élément `price`. La troisième ligne, nous oblige (`#REQUIRED`) à avoir un attribut pour l'élément `book`, appelé `year`. `#REQUIRED` impose la présence d'un attribut pour une balise. Il existe aussi la possibilité de mettre des clefs[1] dans les attributs avec la valeur `#ID` (`#IDREF` pour une clef étrangère), et la possibilité d'utiliser des énumérations. Une DTD peut spécifier le type des valeurs comme du `CDATA` ou `PCDATA` correspondant à du texte, mais ne peut pas, par exemple, spécifier les valeurs autorisées des champs ou paramètres, ce en quoi elle se distingue de XML Schema et Relax NG [rel]. De plus, la norme DTD fait appel à une syntaxe spécifique distincte de XML.

2.1.2.2 XML Schema

XML Schema est un langage de description de format de documents XML, spécifié par le W3C [W3C01b] permettant de définir la structure d'un document XML, par un fichier de description de schéma (*XML Schema Description*, ou fichier XSD) qui est lui-même un document XML.

XML Schema est une alternative aux DTD, et est leur successeur possible, car les DTD sont celles qui sont le plus utilisées. Dans une DTD, on spécifie les étiquettes d'éléments autorisés dans le document, et pour chaque étiquette, on indique les attributs optionnels ou obligatoire, et on contraint le contenu par une expression régulière. Dans une spécification XML Schema, le modèle de contenu ne dépend pas seulement de l'étiquette de l'élément, mais aussi de sa position dans l'arbre ; cela permet d'exprimer des contraintes plus fines. XML Schema possède de plus une notion de «type simple», pour spécifier la forme des valeurs atomiques dans les documents et la manière de les interpréter (comme un entier, une date,...).

Il est possible de définir en plus par rapport aux DTD, des valeurs par défaut des éléments et attributs, la cardinalité d'un élément (grâce aux valeurs d'intervalles définis par `minOccurs` et `maxOccurs`), des intervalles de valeurs, la conformité par rapport à un motif (par exemple : `<xs :pattern value="([a-z])*"/>` pour déclarer une chaîne de caractères en caractères minuscules). Dans l'exemple de

[1] Dans les bases de données, une clef permet d'identifier de manière unique des enregistrements dans une table. Une clef étrangère est une clef qui fait référence à une clef d'une autre table.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="book">
 <xsd:complexType>
  <xs:attribute name="year" type="xs:positiveInteger"/>
  <xsd:sequence>
   <xsd:element name="title" minOccurs="1" maxOccurs="1" type="xsd:string"/>
   <xsd:element name="author" minOccurs="1" maxOccurs="unbounded"/>
    <xsd:complexType>
     <xs:sequence>
      <xs:element name="last" minOccurs="1" maxOccurs="1" type="xs:string"/>
      <xs:element name="first" minOccurs="1" maxOccurs="1" type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
</xsd:schema>
```

FIG. 2.4 – Une spécification XML SCHEMA

spécification XML SCHEMA de la figure 2.4, on définit un élément `book` comme étant un type complexe contenant :

- un attribut *year* de type `xs :positiveInteger`,
- une séquence de fils contenant :
  - un unique élément `title` contenant lui-même une chaîne de caractères,
  - un élément complexe `author` présent au moins une fois, et contenant une élément `last` et `first`.

Il existe différents dialectes XML spécifiques. L'intérêt est ainsi de bénéficier de la technologie entourant XML, comme les outils de validation, mais aussi des outils d'analyses syntaxique et sémantique. Le plus connu est XHTML.

2.1.3 XHTML : un format de pages web

HTML (*Hypertext Markup Language*) [W3C99] est le format de données créé et utilisé pour écrire des pages Web.

HTML en tant qu'application de SGML est abandonné au profit de XHTML [W3C02], qui bénéficie de tout l'environnement XML et notamment de la validation. Il existe donc une DTD spécifique à XHTML dont on trouvera la référence en entête d'un fichier XML (à la troisième ligne du fichier XML de la figure 2.5), qui définit les balises que le navigateur comprendra et interprétera, comme la balise <a> qui signifie *anchor* (ancre) et qui permet d'insérer un lien

hypertexte avec l'attribut `href`.

```
<?xml version="1.0" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
   <title>Une page XHTML</title>
 </head>
  <body>
    <p>Ceci est une <u>page web</u> écrite dans le format <b>XHTML</b>.
      La balise <code>&lt;p&gt;</code> signifie la création d'un
      paragraphe.</p>
    <p>Pour vérifier que cette page est bien écrite en <b>XHTML</b>,
    on pourra aller sur ce
      <a href="http://validator.w3.org/">lien</a>.</p>
  </body>
</html>
```

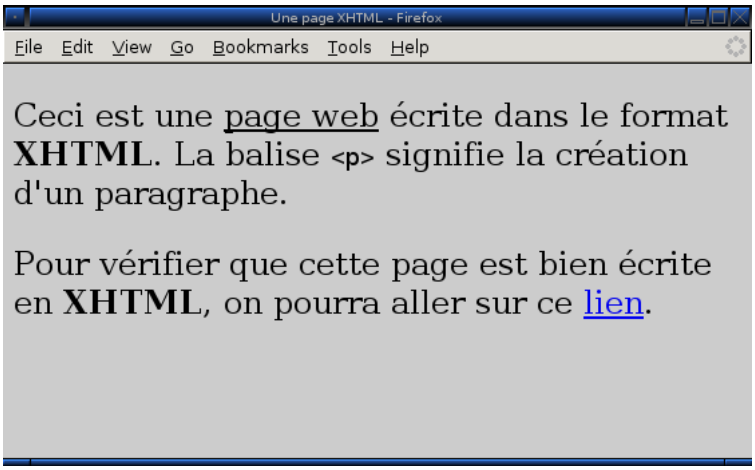


FIG. 2.5 – Une page XHTML et sa représentation dans un navigateur

Il existe des dizaines de dialectes XML spécifiques comme MATHML [W3C01a] pour représenter des figures mathématiques, ou comme SVG (« *Scalable Vector Graphics* ») [W3C04] permettant de décrire des ensembles de graphiques vectoriels,...

2.2 Langages de manipulation de documents XML

Différents langages ont été proposés tant par le W3C que par des équipes de recherche ou encore par l'industrie pour permettre d'extraire de l'information à partir de documents XML, ou pour manipuler des données, de manière, par exemple à n'en garder qu'une partie, ou alors à les combiner avec d'autres données

elles-mêmes extraites à partir d'autres documents.

Il existe dans un langage de manipulation d'arbres XML deux types indissociables de primitives [Cas05] :

- une primitive de déconstruction du document XML, qui permet d'extraire de l'information qu'on appellera *déconstructeur*,
- et la seconde d'itération qui va permettre de répéter l'application d'un déconstructeur particulier sur une collection d'arbres ou sous-arbres XML, dans le but de manipuler ces données et les reconstruire : on l'appellera *itérateur*.

Nous allons exposer les différents déconstructeurs et itérateurs qui existent parmi les langages de manipulation de documents XML.

2.2.1 Déconstructeurs et itérateurs

Il existe deux types de déconstructeurs :

- *les expressions de chemins* qui vont exprimer des contraintes et capturer des nœuds XML par navigation verticale,
- et *le filtrage par motifs* qui opère horizontalement sur une partie du document XML, et qui permet de capturer différents sous-arbres.

La figure 2.6 représente ces deux parcours sur un arbre XML. La représentation sous forme de triangle d'un document XML (image 1) est courante, elle permet de mieux visualiser ce document comme un arbre, où la racine est la cime de ce triangle; les quatre petits triangles représentent quatre sous-arbres contenus dans le document XML, et d'éléments racines <b>, <c>, <d> et <e>. Dans cet exemple, pour capturer ces valeurs XML, le parcours horizontal (image 2) ira depuis la racine jusqu'à leur père respectif (la balise <a>), puis effectuera quatre autres parcours pour capturer chacun de ces sous-arbres séparément. En revanche, le parcours vertical (image 3) descendra niveau par niveau en commençant par la racine, jusqu'à se situer jusqu'au père <a>, puis en un seul motif capturera simultanément les quatre sous-arbres.

Dans la suite, nous présentons XPath qui est le langage de chemins le plus utilisé, et présentons brièvement le filtrage par motifs, qui sera plus détaillé dans la présentation de ℂDuce (section 2.3).

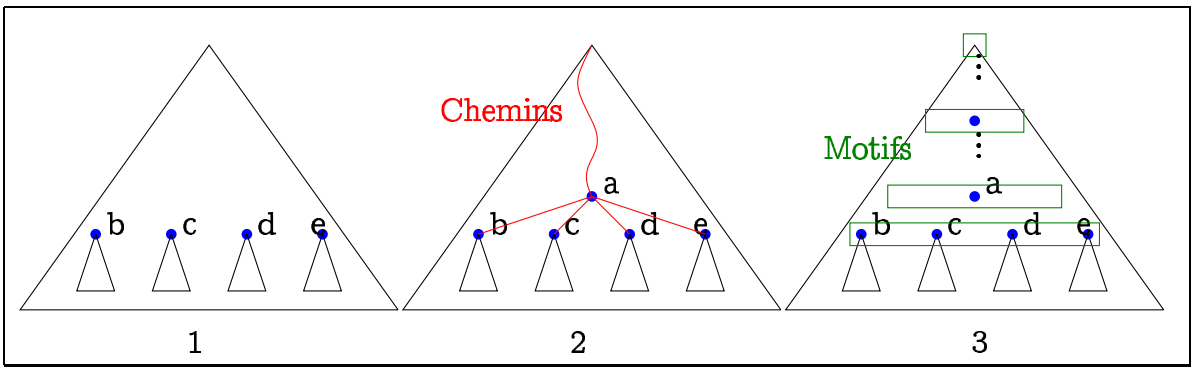


FIG. 2.6 – Représentation des déconstructeurs sur un arbre XML

2.2.1.1 Le parcours vertical

Le plus connu des déconstructeurs par parcours vertical est « *XML Path Language* »(XPATH) [CD99]. XPATH est un langage de chemins qui permet de naviguer à l'intérieur de l'arbre, représentant le document XML. Ce langage permet de désigner un ou plusieurs nœuds d'un arbre, à l'aide d'expressions de chemins. Une expression de chemins est écrite comme une séquence d'étapes pour récupérer à partir d'un nœud XML (appelé nœud contexte), une séquence de nœuds. Chaque étape peut avoir trois composants :

- une *spécification d'axe*, qui permet de spécifier le sens de la recherche entre le nœud contexte et les nœuds à rechercher.
- un *test de nœuds* qui permet de sélectionner le type des nœuds à capturer (élément, texte, attribut,...).
- et des *prédicats* qui permettent de poser des conditions pour sélectionner une partie des nœuds capturés précédemment.

Une étape a pour règle de grammaire [CD99] :

```
[4]   Step          ::=  AxisSpecifier NodeTest Predicate*
                       | AbbreviatedStep
[5]   AxisSpecifier ::=  AxisName '::'
                       | AbbreviatedAxisSpecifier
```

Pour les spécifications d'axe, on dispose de :

- `child` : qui est l'axe par défaut. Cet axe indique que l'on va obtenir les nœuds fils du nœud contexte.
- `descendant` : pour sélectionner les descendants du nœud contexte à n'importe quel niveau.
- `descendant-or-self` : pour sélectionner les descendants du nœud contexte et le nœud contexte.
- `parent` : pour sélectionner le nœud père du nœud contexte.

- `ancestor` : pour sélectionner les ascendants du nœud contexte.
- `ancestor-or-self` : pour sélectionner les ascendants du nœud contexte et le nœud contexte.
- `following-sibling` : pour sélectionner tous les nœuds suivant le nœud contexte.
- `preceding-sibling` : pour sélectionner tous les nœuds précédant le nœud contexte.
- `following` : pour sélectionner tous les nœuds suivant le nœud contexte, à l'exclusion des descendants.
- `preceding` : pour sélectionner tous les nœuds précédant le nœud contexte, à l'exclusion des ancêtres.
- `attribute` : pour sélectionner les attributs du nœud contexte.
- `self` : pour sélectionner le nœud contexte.

Sur l'exemple de la figure 2.1, l'expression XPath `//book/title` sélectionnera tous les titres de la bibliographie. Cette expression va collecter dans un premier temps, à partir de la racine, la séquence des quatre livres, puis pour chaque livre, va sélectionner les éventuelles balises <title>.

Les prédicats (entre crochets) servent à poser des conditions lors d'une étape. Par exemple : `//book[@year='1994']/title/text()` appliqué à la figure 2.1 renverra la chaîne de caractères "TCP/IP Illustrated", correspondant au texte des titres des livres publiés en 1994.

Il existe aussi des fonctions qui peuvent être appliquées à une séquence de nœuds, comme `count()` ou encore `position()` qui donne la position d'un nœud par rapport à ses frères.

XPath est donc un langage de chemins intuitif qui permet de sélectionner une séquence de nœuds dans un document XML. Il est utilisé comme déconstructeur des langages XSLT et XQuery.

Il existe une restriction de XPath qui ne permet pas de remonter aux ascendants et cousins d'un nœud, et qui est appelée la navigation descendante. Les langages XQL et Lorel, entre autres, l'utilisent, et seront présentés dans la suite de ce chapitre.

2.2.1.2 Le parcours horizontal

Le deuxième type de déconstructeur fonctionne par parcours horizontal. Il s'agit du filtrage par motifs (ou « *pattern-matching* ») qui est utilisé en général par les langages fonctionnels pour capturer des sous-valeurs à partir de valeurs de

ces langages. Le filtrage par motifs est une expression de la forme :

```
match e with
    | p1 → e1
    | p2 → e2
    | ...
```

comprenant un opérateur de filtrage `match`, des motifs p_i et des expressions e_i du langage contenant des variables capturées par les motifs p_i. L'opérateur `match` va appliquer à la valeur e, les différents motifs p_i et renverra pour le premier motif qui réussit[2] l'expression e_i correspondante.

Revenons à notre document XML décrivant une bibliographie, et considérons l'exemple :

```
match mon_livre with
   | <book>[ <title>t <author>[<last>l _] _*]]
                          → <resultat> ( t  "a été écrit par" l)</resultat>
   | <book>_ → <resultat>"Ce livre n'a pas de titre ou aucun auteur"</resultat>
   | _ → <resultat>"Ce n'est pas un livre"</resultat>
```

Il s'agit d'un filtrage par motifs sur la valeur `mon_livre`. La syntaxe des motifs est proche de celle de ℂDuce. Un motif réussit si le motif s'applique à la valeur, et capture ainsi les sous-valeurs dans chacune de ses variables. Si aucun motif ne réussit, une erreur est alors renvoyée. Suivant la politique de filtrage choisie, il y a un ordre de priorité à considérer sur les motifs à évaluer.

Cette expression va, dans ℂDuce, évaluer chaque motif l'un après l'autre. Si le premier motif réussit, cela signifie que la valeur `mon_livre` est composée d'une balise `<book>` qui contient un titre et un auteur, qui sont capturés dans les variables t et l ; l'expression de sortie sera évaluée et donnera un nouvel élément `resulat` contenant la chaîne de caractères du titre concaténée avec la chaîne de caractères "a été écrit par" et la chaîne de caractères du nom de famille du premier auteur. Dans le cas où la valeur `mon_livre` contient une balise `<book>` mais ne contient pas d'abord un titre puis un auteur, alors le premier motif échoue. Dans ce cas, le deuxième motif est évalué. Le joker (noté par le caractère "_") permet d'accepter n'importe quelle valeur. Le deuxième motif va filtrer la valeur `mon_livre` si elle contient une balise `<book>`, et ne pas explorer son contenu. Si ce motif réussit, la valeur `<resultat>"Ce livre n'a pas de titre ou aucun auteur"</resultat>` sera renvoyée. Le dernier motif permet de s'assurer que ce filtrage par motif renverra une valeur, en acceptant toutes les valeurs possibles. Dans cet exemple le

[2]Dans le cas d'une politique de filtrage « *first match policy* » ; on y reviendra dans la partie sur ℂDuce.

filtrage sera fait sur ce dernier motif uniquement si la variable `mon_livre` n'a pas pour balise racine `<book>`.

Cet exemple illustre la capture (par les variables t et l) de deux sous-arbres distincts en un seul passage, contrairement à XPath.

Le filtrage par motifs et la navigation XPath sont utilisés par différents itérateurs dans différents langages de manipulation de documents XML. Ils sont présentés dans le tableau synthétique de la figure 2.7.

Langage	Itérateur	Déconstructeur	Typage
SgmlQL	`select from where`	motifs	Oui
Lorel	`select from where`	navigation	Oui
XML-QL	`where in construct`	motifs + exp. reg. de chemins	Non
XSLT	règles	XPath	Non
XQuery	FLWR	XPath	Oui
TQL[CGA+02]	`from select`	motifs	Oui
XDuce	`filter`	motifs	Oui
Cω	`select from where`	navigation	Oui
loto-ql[PV00]	`select where`	motifs	Oui
Xtatic	`iterate`	motifs	Oui
XLinq	`from where orderby select`	XPath	Oui
$\mathbb{C}$Duce	`transform`	motifs	Oui

Fig. 2.7 – Tableau récapitulatif des itérateurs et déconstructeurs de différents langages de manipulation de documents XML

Une fois que l'on dispose d'un déconstructeur nous permettant de capturer des motifs, il faut recourir à un itérateur afin d'appliquer ce déconstructeur. Différents langages existent aussi bien des langages de requêtes que des langages de programmation moins déclaratifs. Nous allons faire une présentation rapide pour certains et présenter ensuite de manière plus complète $\mathbb{C}$Duce (qui servira de support à ce travail de thèse) en particulier nous détaillerons les notions de typage, de sous-typage et de filtrage par motifs.

Les deux premiers langages que nous présentons sont Lorel et SgmlQL [LMR95] qui sont des extensions de OQL [Clu98, BDK92] qui est le standard ODMG en terme de langages de requêtes pour bases de données orientées objet.

2.2.2 LOREL et SGMLQL

LOREL [AQM+97] est un langage de requêtes pour les données semi-structurées. LOREL relaxe le typage inapproprié d'OQL (car trop rigide) pour ce type de données, et utilise comme déconstructeur l'accès par navigation descendante à travers un arbre, avec un itérateur à la SQL, le `select from where`. Un exemple de requêtes qui permet de capturer tous les auteurs du livre ayant pour titre "TCP/IP Illustrated" de la bibliographie `Bib` est :

```
select X.author
from Bib.book X , X.title Y
where Y = "TCP/IP Illustrated"
```

L'itérateur est le `select from where` qui va itérer sur une collection de valeurs capturées par le déconstructeur par navigation. L'expression de chemins `Bib.book.title` donnera l'ensemble de titres des livres de la bibliographie `Bib`. Le déconstructeur est une expression de chemins qui utilise des expressions régulières (intersection, union, option, étoile de Kleene,...) mais ne permet pas de remonter dans l'arbre ni même de poser des conditions directement dans le chemin comme le propose XPATH.

Contrairement à LOREL, SGMLQL [LMR95] est muni du filtrage par motifs comme déconstructeur. La requête ci-dessus s'exprimera ainsi :

```
select a
from <book> title(t), author*(a), ... </book> in bib
where t = "TCP Illustrated"
```

Cette requête va filtrer chaque livre de la bibliographie avec le motif `<book> title(t), author*(a), ... </book>` qui signifie que si un livre a un premier fils qui a pour balise `<title>`, et si à partir du deuxième fils il existe une suite, éventuellement nulle, de balises `<author>` alors on capture dans la variable `t`, le titre, et dans la variable `a` les auteurs. L'algèbre de motifs est donnée dans la figure 2.8.

La différence entre ces deux langages est le déconstructeur. La navigation par chemins de LOREL est plus intuitive, mais en revanche le filtrage par motifs permet de capturer en un même motif différents éléments liés à un même niveau dans l'arbre comme ici, les auteurs et le titre (qui nécessiteraient deux expressions de chemins).

p	: :=	_	joker
		T	type
		$<>\ p\ </>$	capture du contenu
		$<$T$>\ p\ </>$	capture du contenu d'une balise de type T
		#PCDATA	chaînes de caractères
		'reg'	où reg est une expression régulière UNIX
		(p)	équivalent à p
		p*	capture de la liste dont tous les éléments sont filtrés par p
		p+	équivalent à p,p*
		p ?	p ou liste vide
		$p1$, $p2$	concaténation

FIG. 2.8 – Algèbre de motifs de SGMLQL

2.2.3 XML-QL

XML-QL [DFF+98, DFF+99] est un langage de requêtes pour XML qui utilise le filtrage par motifs à l'intérieur d'une expression `WHERE IN CONSTRUCT`, qui va itérer sur une valeur XML capturée dans le `IN` et lui appliquer un motif présent dans le `WHERE`, et ensuite reconstruire une valeur XML dans la partie `CONSTRUCT`. Par exemple, cette expression va grouper dans une balise `<result>` tous les auteurs et titres des livres publiés par **Addison-Wesley**.

```
WHERE <book>
         <publisher>Addison-Wesley</>
         <title> $t</>
         <author> $a</>
      </>
IN "bib.xml"
CONSTRUCT <result>
             <author> $a</>
             <title> $t</>
          </>
```

En un seul motif, on teste le nom de l'éditeur, et on capture le titre et l'auteur correspondants dans les variables `$t` et `$a`.

L'algèbre de motifs de XML-QL contient l'alternative (|), la concaténation (.) (qui est utilisée pour la navigation descendante), l'étoile de Kleene et le + dans les expressions régulières qui portent sur les nœuds, ainsi que la capture de variables pour récupérer le contenu d'un élément.

La particularité de XML-QL tient dans l'utilisation d'un autre déconstructeur : les expressions régulières de chemins.

```
WHERE <part+.(subpart|component.piece)>$r</>
IN "parts.xml"
CONSTRUCT <result> $r</>
```

Pour illustrer ces expressions régulières de chemins, regardons la requête ci-dessus. L'expression `part+` correspond à `part.part*`, c'est-à-dire qu'elle va filtrer récursivement les enfants <part> d'éléments <part>, (donc tous les descendants tels que chaque parent soit un élément `part`), et ensuite pour chacun d'eux appliquer le motif `(subpart|component.piece)` qui va capturer les élément fils <subpart> et les enfants <piece> contenus dans <component>.

Ces expressions régulières de chemins offrent un déconstructeur proche de XPath. XML-QL propose donc à l'intérieur d'une requête, l'utilisation de deux types de déconstructeurs. Néanmoins, ce langage n'est pas typé, et, nous le verrons dans la section suivante, n'a pas la même expressivité que les motifs ℂDuce qui permettent, par exemple, d'exprimer la négation, ou l'intersection dans les expressions régulières (et, nous le verrons par la suite, dans les projections qui représentent une navigation descendante ; par exemple, la figure 3.6 à la page 60).

2.2.4 XQuery

[CFMR03] définit les caractéristiques et fonctionnalités attendues d'un langage de requêtes pour XML. Les requêtes doivent aussi bien pouvoir être opérées sur un simple document XML que sur une collection de documents XML.

Elles doivent pouvoir sélectionner les documents entiers, ou au contraire ne garder que des sous-parties de ces documents qui remplissent certaines conditions ; ces conditions pouvant porter sur le contenu ou sur la structure. Parmi ces objectifs, on retrouve :

- l'importance de la déclarativité pour une requête, avec le découpage de la requête en trois parties.
- la présence de quantificateurs existentiel et universel.
- la combinaison d'informations de différentes parties d'un document, ou de plusieurs documents.
- l'agrégation d'informations à partir de documents proches.
- le tri sur les éléments.
- l'imbrication de requêtes.

XQuery [BCF+01] est issu de ces recommandations. Le langage a été conçu pour permettre d'exprimer des requêtes facilement compréhensibles : les FLWR (prononcer « flower »), qui n'est rien d'autre que l'itérateur de XQuery. XQuery

est un dérivé de QUILT [CRF01], empruntant de nombreuses fonctionnalités de XPATH, XQL [RLS98], XML-QL [DFF+98], SQL et OQL. XQUERY est un langage de programmation qui utilise comme primitive de déconstruction les expressions XPATH et comme itérateur le `for where return`.

Les FLWR sont composées de :

- `for` : qui est un mécanisme d'itération sur un ensemble de nœuds
- `let` : qui permet l'assignation de variables
- `where` : comme les clauses for et let génèrent un ensemble de nœuds, il peut être utile de les filtrer par des conditions
- et `return` : qui construit le résultat pour chaque nœud vérifiant la clause where

Avec la possibilité de construction de nouvel élément.

La figure 2.9 représente une requête XQUERY qui s'applique à l'exemple de la figure 2.1, et qui est la première requête des « *XML Query Use Cases* » [CFF+03]. Cette requête construit dans un nouvel élément `book`, les titres des livres de la bibliographie se trouvant à l'adresse `http://bstore1.example.com/bib.xml` et leur année, tels que l'éditeur soit "Addison-Wesley" et que son année soit postérieure à 1991, et en les encapsulant dans l'élément `bib`. Il s'agit d'une sélection (sur l'année et l'éditeur), et d'une projection (en retirant les éléments fils de `book` : `author` ou `editor`, `publisher` et `price`).

```
<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. 2.9 – Requête Q1 des « *XML Query Use Cases* »

XQUERY est un langage fortement et statiquement typé [DFM+02]. On peut y définir des types mutuellement récursifs, ainsi que des fonctions récursives. Son système de types est fondé sur celui de XDUCE [HVP05].

2.2.5 Cω et XLinq

Cω [BMS05] est une extension de C# [csh05] qui lui apporte un typage fort, un itérateur (`select from where`) et qui utilise un langage de navigation descendante de chemins. Un exemple de requête Cω :

```
SELECT <book>
         b.title
         a.author
      </book>
FROM book b in Bib.book
WHERE b.publisher == <publisher>"Addison-Wesley"
```

XLinq [xli, MB05] a été conçu par Microsoft pour .NET et fait partie du projet LINQ [BH05] consistant à offrir des langages de requêtes typés intégrés à C# [csh05] qui possède un langage de requêtes qui n'est pas typé. Il sera ainsi possible d'effectuer des requêtes typées sur des bases de données, des documents XML,...
XLinq est l'extension de LINQ qui permet d'exécuter des requêtes sur des documents XML, et de créer ou de transformer du XML. L'itérateur proposé est proche du FLWR de XQuery, ainsi que le déconstructeur qui est proche de XPath.

2.2.6 XDuce

XDuce (qui se prononce « transduce ») a été défini et implanté dans la thèse de Haruo Hosoya [Hos01]. XDuce [HP03, HVP00] est un langage de programmation fonctionnel typé adapté à la manipulation de documents XML. Son système de types repose sur une algèbre d'expressions régulières de types. Le langage propose notamment un itérateur de filtrage par motifs [HP01], et un déconstructeur reposant sur une algèbre de motifs adaptée à l'extraction de données. Les types sont interprétés comme des ensembles de valeurs, où une valeur représente un arbre XML ou une feuille. Il faut noter que dans les langages fonctionnels, une valeur du langage représente un élément XML, et contient à la fois son contenu et la position dans le document source. La syntaxe de XDuce est relativement proche de XML, ainsi une partie du document XML de référence peut être représentée par la valeur `mybib` :

```
let val mybib =
  bib[
    book[
      title["TCP/IP illustrated"],
      author[ last["Stevens"] , first["W."] ],
      publisher["Addison-Wesley"]
    ]
...
]
```

XDUCE possède une algèbre de types et de motifs [HP01] qui permet d'utiliser le filtrage par motifs, qui est donnée dans la figure 2.10. Dans les motifs, on compte les déclarations de variables x, la séquence vide `()`, des noms de motifs y, un nœud l contenant une séquence sur laquelle un motif est appliqué $l[p]$, la concaténation représentée par la virgule, et l'alternative. On dispose pour la création de types de la séquence vide, des noms de types définis (`T`), des nœuds contenant une séquence, de la concaténation de types, et de l'alternative de types. Il existe aussi la fermeture transitive $t*$ d'un type t, qui est définie par le type X de manière récursive : `type X =` t`,X|().` `T?` et `T+` dont définis de la même manière.

$$p \quad ::= \quad x \mid x \text{ as } P \mid () \mid y \mid l[p] \mid p,p \mid p|p$$

algèbre de motifs

$$t \quad ::= \quad () \mid \mathtt{T} \mid l[t] \mid t|t \mid t,t$$

algèbre de types

FIG. 2.10 – Algèbre de motifs et de types de XDUCE

Par exemple, considérons les déclarations de types suivantes avec l'optionnalité sur le nombre d'auteurs dans le type `Book` :

```
type Book = book[(Title,Author?,Publisher)]
type Title = title[String]
type Author = author[(Last,First)]
type Last = last[String]
type First = first [ String]
type Publisher = publisher [String]
```

Dans l'exemple suivant, nous définissons une valeur `books` qui récupère l'ensemble des livres de la bibliographie, avec l'application de l'itérateur `filter` qui

filtre sur l'unique valeur `mybib`, le motif `bib[ val bk ]`, et retourne la valeur `bk`.

```
let val books =
  filter mybib {
    bib[ val bk ]{ bk }
  }
```

XDuce possède la possibilité de définir des fonctions; dans l'exemple ci-dessous est définie la fonction `mkBiblio` qui renvoie une séquence composée d'éléments `<entry>` contenant chacun des titres des livres qui a au moins un auteur. Celle-ci prend en argument une valeur dénotée par `bk` de type `Book*` et renvoie une valeur de type `entry[Title]*`. Le `filter` va appliquer les deux motifs, le premier (qui vérifie s'il y a un auteur) pour capturer le titre, et le second qui renvoie la séquence vide, dans le cas où le livre n'a pas d'auteur.

```
fun mkBiblio (val bk as Book* ) : entry[Title]* =
  filter bk { book[Title,Publisher]*,
              book[title[val t],author[last[val l],first[val f]],
              publisher[val p] ],
              val rest
            { entry[title[t]], mkBiblio(rest) }
        | book[Title,Publisher]*
           { () }
  }
let val _ = print(mkBiblio(books))
```

XTatic [GLPS05] a été conçu comme une extension de C# avec un système de types statique pour la manipulation de documents XML, et utilise les expressions régulières de types et le filtrage par motifs de XDuce.

Nous allons maintenant détailler le langage ℂDuce qui a servi de base pour la définition du langage ℂQL. Nous illustrerons le filtrage par motifs en détail, ainsi que l'intérêt qu'apporte son inférence de types.

2.3 ℂDuce

ℂDuce[3] [FCB02, BCF03, Fri04a, Fri04c] est un langage de programmation fonctionnel et typé qui a été conçu pour manipuler des documents XML. Il

[3]Une documentation détaillée est disponible sur `www.cduce.org`, ainsi que les sources et un prototype.

possède certains aspects classiques pour un langage fonctionnel comme l'ordre supérieur, d'autres aspects plus rares comme les fonctions de surcharge dynamique, et d'autres plus spécifiques à XML, comme le support des caractères Unicode, des espaces de noms (« *Namespaces* »), des DTD et une partie de XML SCHEMA.

2.3.1 Algèbre de types

L'algèbre de types de ℂDuce possède trois familles de types scalaires :

- les entiers, qui sont représentés soit par le type général qui représente les entiers (`Int`), soit par le type intervalle de la forme `i--j` (où `i` et `j` sont des entiers simples) ou soit par des entiers simples comme 42 qui représente le type singleton ne contenant que la valeur 42.
- les caractères, qui sont représentés soit par le type général qui représente les caractères (`Char` l'ensemble des caractères Unicode) ou par le type `Byte` (l'ensemble des caractères Latin1), soit par le type intervalle de la forme `c--d` (où `c` et `d` sont des caractères simples de la forme `'a'`, `'b'`, ..., ou des caractères spéciaux) ou par des caractères simples correspondant aux types singleton.
- les atomes, qui sont des constantes prédéfinies (comme `` `nil ``, `` `true ``, ...) ou qu'un utilisateur peut définir. Ces types atomes qui sont eux-mêmes des types singletons appartiennent au type général `Atom` (qui sont précédés par une quote simple fermante).

Les autres types de l'algèbre de ℂDuce sont définis à partir de ces types scalaires (de manière éventuellement récursive), des types `Any` et `Empty` (correspondant respectivement aux types universel et vide), et par l'application de *constructeurs* et de *combinateurs* de types.

Combinateurs de types ℂDuce possède l'ensemble des combinateurs booléens. Si t_1 et t_2 sont des types, alors $t_1 \& t_2$ est le type correspondant à l'intersection de ces deux types, $t_1 \mid t_2$ leur union, et $t_1 \setminus t_2$ leur différence. Par exemple, le type `Bool` est défini dans ℂDuce comme l'union des deux types singletons contenant les atomes vrai et faux, qui est le type `` `true | `false ``.

Constructeurs de types ℂDuce offre le constructeur de types pour les types enregistrements $\{ a_1 = t_1 ; \ldots ; a_n = t_n \}$, pour les types produits (t_1, t_2), et pour les types fonctionnels $(t_1 \rightarrow t_2)$. Les constructeurs qui nous intéresseront le plus sont ceux qui correspondent à la création de séquences et de valeurs XML.

Les types séquences sont représentés par des crochets qui entourent une expression régulière de type. Ainsi par exemple, les chaînes de caractères qui sont une séquence de caractères seront représentées par le type `String` qui est lui-même défini par `[ Char* ]`. ℂDuce utilise la syntaxe standard pour les expressions régulières[4]

$$r ::= t \quad r\,r \quad r|r \quad r* \quad r+ \quad r? \quad \varepsilon$$

qui est une manière simple et très proche de XML pour exprimer certains types récursifs.

La forme générale des types XML est $<t_1\ t_2>t_3$ où les t_i sont des types. Dans le cadre de XML, t_1 est toujours un type singleton contenant le type atome (correspondant au nom) de l'étiquette, t_2 est un type enregistrement (qui contient un ensemble d'attributs), et t_3 un type séquence (pour les fils de cet élément). Un atome en ℂDuce commence par une quote simple fermante. ℂDuce autorise son oubli par souci de confort syntaxique, ainsi que l'oubli des accolades et des points-virgules pour les attributs, ce qui fait qu'une valeur XML aura cette forme (où r est une expression régulière de type) :

$$<tag\ \ a_1{=}t_1\ \ a_2{=}t_2\ \ \ldots\ \ a_n{=}t_n>[\ r\ ]$$

Dans la figure 2.11, la DTD de la figure 2.3 est présentée ainsi que les déclarations de types de ℂDuce qui lui correspondent. L'utilisation de l'expression régulière `PCDATA` est une autre convention pour représenter l'expression régulière `Char*`.

2.3.1.0.1 Sous-typage Les types sont définis par une fonction d'interprétation ensembliste $[\![_]\!] : \mathcal{I} \to \mathcal{P}(\mathcal{D})$, où $\mathcal{I}$ est l'ensemble des types et $\mathcal{D}$ un domaine. $[\![_]\!]$ est défini par :

$$\begin{array}{rcl} [\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\ [\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\ [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \\ [\![\neg t]\!] & = & \mathcal{D} \setminus t \\ [\![\texttt{Any}]\!] & = & \mathcal{D} \\ [\![\texttt{Empty}]\!] & = & \varnothing \end{array}$$

La définition exacte est donnée très précisément dans [FCB05].

Avec cette présentation ensembliste des types, la relation de sous-typage devient alors évidente :

[4] Il existe aussi une version « gloutonne »pour certains opérateurs postfixes d'expressions régulières [BCF03].

DTD :

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

Types $\mathbb{C}$Duce :

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title (Author+ | Editor+ ) Publisher Price]
type Author = <author>[Last First]
type Editor = <editor>[Last First Affiliation]
type Title  = <title>[PCDATA]
type Last  = <last>[PCDATA]
type First  = <first>[PCDATA]
type Affiliation = <affiliation>[PCDATA]
type Publisher  = <publisher>[PCDATA]
type Price  = <price>[PCDATA]
```

FIG. 2.11 – DTD et types $\mathbb{C}$Duce types représentant une bibliographie

$t \leq s$ si et seulement si $[\![t]\!] \subseteq [\![s]\!]$

qui veut dire qu'un type t sera un sous-type de s, si et seulement si l'ensemble des valeurs ($[\![t]\!]$) de type t, est inclus dans l'ensemble des valeurs ($[\![s]\!]$) de type s. Par exemple, comme le type $Bool$ est égal à l'ensemble {`true,`false}, alors le type $true$ (contenant uniquement la valeur `true) est un sous-type du type $Bool$.

Expressions

Les constructeurs d'expressions suivent le plus souvent la même syntaxe que leur type correspondant ; par exemple une expression représentant un enregistrement a pour forme : $\{ a_1 = e_1 ; \ldots ; a_n = e_n \}$, et une expression représentant une paire : (e_1, e_2). La convention d'écriture sur les types XML s'applique ici de la même manière ; au lieu d'écrire `<`book year="1990">[...]`, nous pourrons écrire `<book year="1990">[...]`. Ainsi, le document XML de référence présenté dans la figure 2.1, sera représenté par l'expression $\mathbb{C}$Duce de la figure 2.12 validés chacun respectivement par la DTD et par le type `Bib` de la figure 2.11.

```
<bib>[
        <book year="1994">[
                <title>['TCP/IP Illustrated']
                <author>[<last>['Stevens']
                         <first>['W.']]
                <publisher>['Addison-Wesley']
                <price>['65.95']
        ]
        <book year="1992">[
                <title>['Advanced Programming in the Unix environment']
                <author>[<last>['Stevens'] <first>['W.']]
                <publisher>['Addison-Wesley']
                <price>['65.95']
        ]
        <book year="2000">[
                <title>['Data on the Web']
                <author>[<last>['Abiteboul'] <first>['Serge']]
                <author>[<last>['Buneman'] <first>['Peter']]
                <author>[<last>['Suciu'] <first>['Dan']]
                <publisher>['Morgan Kaufmann Publishers']
                <price>['39.95']
       ]
]
```

FIG. 2.12 – Une valeur $\mathbb{C}$Duce représentant le document XML de référence

Comme il existe des constructeurs d'expressions, il existe aussi des destructeurs. Par exemple, pour les enregistrements l'expression $r \setminus \ell$ retire les attributs ℓ , s'ils existent, de l'ensemble d'attributs r ; r+r' représente l'addition destructive des attributs de r' de ceux de r. Les opérateurs qui vont le plus nous intéresser le long de ce travail sont ceux qui manipulent des séquences. Il existe l'opérateur infixe `@` qui correspond à la concaténation de deux séquences, l'opérateur préfixe `flatten` qui prend des séquences de séquences et retourne leur concaténation, et le plus important qui est l'opérateur `transform`. Sa syntaxe est :

```
transform e with
      p1 -> e1
    | p2 -> e2
    | ...
    | pn -> en
```

avec $n \geq 1$ et où e, e_1, e_2, ..., e_n sont des (expressions qui retournent des) séquences et p_1, p_2, ..., p_n sont des *motifs*.

Cette expression parcourt la séquence e et filtre tous les éléments de e avec les motifs p_1, p_2, ..., p_n, tout en suivant une « *first match policy* » . Il s'agit

d'une politique de priorité sur le premier filtre ce qui signifie que si un élément ne peut pas être filtré par p_1, alors il va être filtré par le suivant p_2. S'il ne peut non plus être filtré, on va essayer le suivant, et ainsi de suite. Dès qu'un élément de e est filtré par un motif p_i, alors l'expression e_i correspondante est évaluée dans un nouvel environnement où les variables du motif sont substituées par les valeurs capturées par ce motif. Si une valeur n'a pu être filtrée par un motif, alors la valeur représentant la séquence vide est renvoyée. Quand tous les éléments de la séquence e ont été parcourus et filtrés, l'opérateur `transform` renvoie la concaténation de tous les résultats.

2.3.2 Filtrage par motifs

Les motifs les plus simples sont les types eux-mêmes et les variables. Un motif ne contenant qu'une variable `x` réussit toujours et capture en `x` la valeur sur laquelle ce motif est filtré.

Si e est une séquence d'entiers alors l'expression

```
transform e with x -> if x>=0 then [x] else [ ]
```

renverra la sous-séquence de e qui contiendra uniquement ses entiers positifs. Un motif ne contenant que le type t réussira uniquement s'il est filtré par une valeur de type t, ou d'un sous-type de t. On peut écrire des motifs plus expressifs en utilisant les constructeurs de type et les combinateurs "`&`" et "`|`". Ainsi p_1`&`p_2 réussira seulement si p_1 et p_2 réussissent (p_1 et p_2 peuvent avoir des variables de capture différentes), tandis que p_1`|`p_2 (qu'on appellera « motif alternatif ») réussira si p_1 réussit, ou si p_1 échoue et p_2 réussit (une restriction est donnée telle que p_1 et p_2 doivent avoir les mêmes variables de capture).

Par exemple, le motif `x&Int` réussira uniquement s'il est filtré par un entier, et dans ce cas, la variable de capture `x` sera ensuite liée à cet entier. Comme le type représentant les seuls entiers positifs peut être exprimé par le type intervalle[5] `0--*`, alors l'expression précédente pourra être écrite en utilisant un motif plus fin :

```
transform e with x&(0--*) -> [x]
```

L'expression suivante montre un cas d'utilisation de plusieurs motifs, qui permet de transformer les entiers négatifs en entiers positifs :

[5] Dans les intervalles sur les entiers, `*` correspond à l'infini. `*--0` dénotera les entiers négatifs

```
transform e with x&(0--*) -> [x] | x&(*--0) -> [-x]
```

Si nous savons que e est une séquence formée que d'entiers alors nous pouvons retirer "`&(*--0)`" dans le second motif car cela constitue de l'information redondante. En effet, une valeur ne pourra être filtrée par ce second motif uniquement si cette valeur est un entier négatif (en réalité ℂDuce retire ces informations redondantes à la compilation), mais si e est une séquence ne contenant pas que des entiers alors le « type checker »(validateur de types) retournera une erreur de type.

Les motifs construits avec des constructeurs de type sont également simples. Par exemple, le motif[6] appliqué à une bibliographie :

```
<book year=y>[ <title>t <author>[ _ f ] ;_ ]
```

va filtrer toutes les valeurs représentant des livres et va lier à `y` la valeur de l'attribut `year`, à `t` la chaîne de caractères du titre, et à `f` l'élément `<first>` (et son contenu) du premier auteur. Le joker (`_`) est utilisé dans les motifs comme un raccourci du type universel `Any` (qui filtre n'importe quelle valeur, dans notre cas cela filtre l'élément `<last>`) et "`;_`" pour filtrer toute la fin de la séquence (les auteurs éventuels suivants, l'éditeur et le prix, et non pas le rédacteur puisque le motif impose la présence d'un auteur).

Le filtrage par motifs est aussi fortement utilisé dans la définition de fonctions, que ce soit dans la partie des paramètres (pour décomposer les arguments) que dans le corps de la fonction. Par exemple :

```
fun authors (<bib>books : Bib) : [String*] =
transform books with
| <book year=("1999"|"2000")>[ _ <author>[ _ <first> f ] ;_ ] -> [ f ]
| <_ ..>[_ <_>[<_>s ;_] ;_ ] -> [s]
```

`authors` une fonction (de type `Bib` $\rightarrow$`[String*]`) qui va parcourir la séquence des éléments de l'argument (lié à la variable `books` par le motif `<bib>books` en paramètre) et pour chaque livre, va retourner la chaîne de caractères concaténée avec les chaînes de caractères des prénoms du premier auteur pour les livres ayant été publiés en 1999 ou 2000, ou avec les chaînes de caractères correspondant au nom de famille du premier auteur pour les livres publiés dans une autre année. On peut remarquer que le second motif est très obscur car utilisant un grand nombre de jokers, mais pour autant il filtre bien n'importe quel livre (il suffira d'essayer de remplacer les jokers par les étiquettes correspondantes pour s'en assurer).

[6] Pour les motifs XML, la même convention pour les types et les expressions s'applique.

Il existe aussi d'autres constructeurs dans les motifs qui sont les captures par défaut.

Les captures par défaut dans un motif sont de la forme `(x :=`e`)` ; le motif réussira à chaque fois et liera la valeur de e à `x`. Ils sont utiles dans les motifs alternatifs qui imposent d'avoir les mêmes variables pour assigner une valeur par défaut à une variable de capture qui n'était pas filtrée dans les alternatives précédentes d'un même motif.

Par exemple dans la fonction `authors` précédente, dans le cas où nous voudrions garder dans les livres n'ayant pas été publiés en 1999 ou en 2000, non plus le nom de famille du premier auteur, mais celui du second. Il est possible qu'il n'y ait qu'un seul auteur, et dans ce cas, le motif échouera. Grâce à la capture par défaut, nous pourrions écrire :

```
transform books with
| <book year=("1999"|"2000")>[ _ <author>[ _ <first> f ];_ ] -> [ f ]
| <_ ..>[ _ <author>_ ( <author>[<first>s _] | (s :="none") );_ ] -> [ s ]
```

Le second motif est ainsi modifié avec la capture par défaut à l'emplacement du deuxième auteur (le motif est plus explicite uniquement pour une question de lisibilité, mais ces informations redondantes sont retirées à la compilation).

Et enfin le dernier constructeur est la capture de séquence dans une variable qui est de la forme $x :: R$ où R est une expression régulière de type ; le motif va lier x à la *séquence* d'éléments filtrés par l'expression régulière R. Ces motifs sont utiles pour capturer une sous-séquence entière dans une séquence.

Par exemple dans notre fonction `authors` qui retourne pour chaque livre de la bibliographie la liste des auteurs se trouvant dans un élément `<authors>`, nous pourrions écrire :

```
fun authors (<bib>books : Bib) : [ (<authors>[Author+])* ] =
transform books with
<book>[ _ (a : :Author+);_ ] -> [ <authors>a ]
```

Cette fonction appliquée à une bibliographie nous donnera la séquence composée de tous ses auteurs et qui sera de type `[ (<authors>[Author+])* ]`.

Il existe une importante différence entre `[ x::Int ]` et `[ (x & Int) ]`. Les deux acceptent une séquence formée par une seul entier mais dans le premier cas `x` sera lié à une séquence (ne contenant qu'un seul entier) alors que dans le second, il sera lié à l'entier lui-même.

Dans le cas de la fonction précédente le système de types de ℂDuce est capable de détecter que si la bibliographie est non vide, alors la fonction `authors` retournera au moins un auteur, ce qui est indiqué par le + de l'expression régulière signifiant « au moins un », mais que comme la bibliographie peut ne pas contenir de livres, alors cette fonction pourra ne pas renvoyer de séquences de type `<authors>[Author+]`, ce qui est indiqué par l'étoile de Kleene (∗) signifiant « possiblement vide ».

type de e	type inféré pour `transform`
`[Int String Int]`	`[Int Int]`
`[Int | String]`	`[Int?]`
`[Int* String Int]`	`[Int+]`
`[Int+ String Int]`	`[Int+ Int]`
`[(0--10)+ String]`	`[(0--10)+]`
`[(Int String)+]`	`[Int+]`

FIG. 2.13 – Quelques cas d'inférence de types par ℂDuce.

Pour finir et pour montrer la puissance de l'inférence de types de ℂDuce, considérons le motif `[ x::Int*]` et l'expression utilisant ce motif `transform` e `with [ x::Int*] -> [x]`. La figure 2.13 donne les quelques exemples de types inférés pour cette expression en connaissant le type de e.
ℂDuce est capable statiquement de détecter si des motifs ne pourront jamais être filtrés en connaissant le type des expressions à filtrer. Par exemple, si dans la fonction `authors` nous avions fait une faute de frappe dans l'étiquette du livre et écrit `bok` au lieu de `book`, ℂDuce nous le signalera en pointant le fait que la première branche (le premier motif) ne sera jamais utilisée.

Algèbre des types et motifs

La figure 2.14 résume les algèbres de motifs et de type de ℂDuce, où b représente les types simples comme `Char`, `Int`,..., ainsi que les types singletons représentant les valeurs. $t\backslash t$ représente la différence du premier sur le deuxième type. Il existe aussi le type $AnyXml$ qui contient l'ensemble des valeurs XML possibles.

2.4 Conclusion

Nous avons donc présenté dans ce chapitre les différents déconstructeurs et itérateurs qui existent, pour la recherche et l'interrogation de documents XML.

Types

t ::= b | $t|t$ | $t\&t$ | $t\backslash t$ | (t,t) | $t \rightarrow t$
| <t ℓ=t...ℓ=t>t | [R] | `Empty` | `Any`

Expressions régulières de types

R ::= t | $R\,R$ | $R|R$ | $R*$ | $R+$ | $R?$

Motifs

p ::= x | t | $p\&p$ | $p|p$ | (p,p) | <p ℓ=p ... ℓ=p > p | [r]

Expressions régulières de motifs

r ::= p | x : :r | $r|r$ | $r\,r$ | $r+$ | $r*$ | $r?$

FIG. 2.14 – Algèbres des types et motifs de ℂDuce

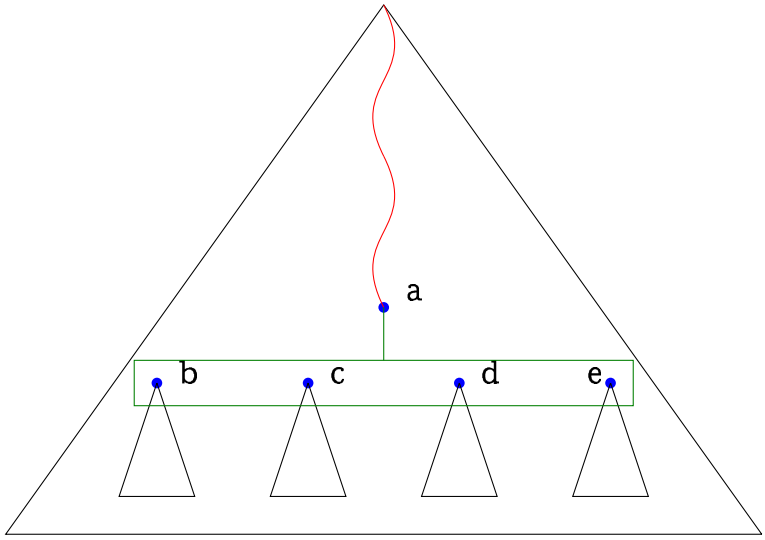


FIG. 2.15 – Représentation d'utilisation conjointe des deux déconstructeurs sur un arbre XML

Nous avons aussi présenté le langage ℂDuce avec son algèbre de types et de motifs, ainsi que le filtrage par motifs.

Le travail de cette thèse va consister en l'apport à ℂDuce d'un langage de requêtes déclaratif qui utilise les deux types de déconstructeurs avec un itérateur à la `select from where`. L'idée principale d'utiliser les deux déconstructeurs pour un seul itérateur est de pouvoir bénéficier de tous leurs atouts. Tandis que la navigation permet de pointer directement les sous-arbres à n'importe quelle profondeur, le filtrage par motifs permet de capturer facilement en un seul parcours différents enfants d'un nœud. En réunissant ces deux aspects présentés dans la figure 2.6, nous pourrions les mélanger et ainsi minimiser le nombre d'expressions, comme présenté dans la figure 2.15, et donc de faciliter l'écriture de requêtes com-

plexes. Un tel langage existe, il s'agit de XML-QL, mais n'est cependant pas un langage typé, et qui n'a pas la même expressivité que les algèbres de motifs et de types de ℂDuce.

Dans le chapitre suivant, nous présenterons et définirons un langage de requêtes qui intègre les expressions de chemins, et le filtrage par motifs de ℂDuce.

Chapitre 3

Le langage de requêtes ℂQL

Sommaire

3.1 Motivations 47
3.2 Description du langage de requêtes 48
3.2.1 Syntaxe 48
3.2.2 Sémantique 53
3.2.3 Exemples de requêtes 57
3.3 Conclusion 64

3.1 Motivations

La nécessité d'avoir un langage de requêtes déclaratif pour ℂDuce qui puisse tirer parti de la puissance expressive de son filtrage par motifs a été mise en évidence dans le *white paper* de ℂDuce[BCF03]. Avoir une interface déclarative permet de spécifier seulement les valeurs à capturer et à calculer, sans obliger un utilisateur à écrire lui-même un algorithme optimal qui puisse le faire ; en laissant ainsi le langage faire ces calculs à sa place. Cette déclarativité permet aussi de poser un cadre fixe qui va permettre au concepteur du langage d'apporter des optimisations, car connaissant les algorithmes qui seront utilisés. C'est essentiellement la différence fondamentale entre un langage de requêtes et un langage de programmation.

Comme on l'a vu dans le chapitre précédent, il existe deux types de déconstructeurs indépendants pour un langage de traitement de documents XML. Nous sommes convaincus qu'apporter ces deux types de déconstructeurs à un langage de requêtes peut aider un utilisateur dans l'écriture de requêtes complexes ou simples.

Ce chapitre présente le langage ℂQL (pour « ℂDuce Query Language ») qui correspond à la première partie de cette thèse. Sa syntaxe et sa sémantique seront

données, avec les deux types de déconstructeurs. Les motifs ℂDuce vont nous permettre de capturer en largeur, et nous proposerons des opérateurs de projection qui simuleront une partie de XPath. Nous présenterons brièvement l'intérêt de ce langage couplé à ℂDuce à travers quelques exemples choisis. Dans les chapitres suivants, nous proposerons des optimisations et étudierons les performances de ℂQL par rapport à d'autres langages de requêtes.

3.2 Description du langage de requêtes

Le langage de requêtes ℂQL est un sous-ensemble de ℂDuce. Son implantation s'est faite dans le cœur de ℂDuce, de manière à profiter du filtrage par motifs, et du sous-typage sémantique. Chaque requête est traduite en du code ℂDuce. Nous présentons dans cette section la syntaxe et la sémantique de ℂQL[BCM04, BCM05], à travers l'écriture de requêtes complexes avec motifs, puis de l'ajout de projections. Ensuite nous présenterons l'expressivité de ℂDuce à travers quelques exemples de requêtes.

3.2.1 Syntaxe

Nous présentons la syntaxe de ℂQL en introduisant un nouvel itérateur (`select from where`) à ℂDuce, qui a pour seul déconstructeur le filtrage par motifs. Ensuite nous ajouterons différentes projections pour simuler la navigation par chemins.

3.2.1.1 Le `select from where` avec motifs

Une requête ℂQL est écrite sous cette forme :

```
select e0
   from p1 in  e1,
        p2 in  e2,
           ⋮
        pn in  en
   where c
```

où les p_i et e_i représentent respectivement des motifs et des expressions ℂDuce, et c une expression booléenne. Une requête se compose en trois parties :

- une clause `select` pour reconstruire le résultat attendu.
- une clause `from` pour spécifier les documents sources et y capturer des sous-documents à partir de motifs, et éventuellement itérer des nouveaux motifs sur des variables capturées précédemment.

– une clause optionnelle `where` pour spécifier des conditions, portant sur des variables, du texte ou des sous-arbres XML, comme des tests d'égalité, d'inégalité, des conditions de jointure,...

La syntaxe formelle de ℂQL est donnée par la grammaire suivante :

Requêtes

q ::= `select` e `from` f `where` c
| `select` e `from` f

Bindings

f ::= p `in` e , f
| p `in` e

Conditions

c ::= `'true`
| `'false`
| `not(` c `)`
| c `or` c
| c `and` c
| `member(` e , e `)`
| e *bop* e *bop* $\in\{=, >, \leq, <, \geq\}$

Expressions

e ::= x		variable
| v		valeur
| `[` $e \ldots e$ `]`		séquençage
| `flatten(`e`)`		aplatissage
| q		requête
| e/t		projection sur élément
| $e/@a$		projection sur attribut
| $e//t$		projection sur descendants
| $\langle\, e \;\; \ell = e \ldots \ell = e \,\rangle\, e$		constructeur XML
| *op*`(` e `)`	*op* $\in\{$distinct_values, count, avg, max, min, sum$\}$	
| $\mathcal{E}$		ensemble des expressions de ℂDuce bien formées sans appel de fonction

Motifs

$p ::= x$		capture
|	t	contrainte de type
|	$p\&p$	conjonction
|	$p \mid p$	alternative
|	(p,p)	paire
|	$\langle\ p\ \ \ell = p \ldots \ell = p\ \rangle\ p$	motif XML
|	$[r]$	séquences
|	$(x := v)$	défaut

Expressions régulières de motifs

$r ::= p$		motif
|	$(x :: r)$	capture de séquence
|	$r \mid r$	alternative
|	$r\ r$	concaténation
|	$r+$	séquence non vide
|	$r*$	séquence
|	$r?$	option

Types

$t ::= B$		types de base et singleton
|	$t \mid t$	union
|	$t\&t$	intersection
|	$t \setminus t$	différence
|	$\langle\ t\ \ \ell = t \ldots \ell = t\ \rangle\ t$	arbre XML
|	$[R]$	Séquence avec expressions régulières de types
|	`Empty`	type vide
|	`Any`	type universel

Expressions régulières de types

$R ::= t$		type
|	$R\ R$	concaténation
|	$R \mid R$	union
|	$R*$	séquence
|	$R+$	séquence non vide
|	$R?$	option

Cette syntaxe nous autorise à utiliser des opérations d'agrégations utiles en SQL, comme l'opération `distinct_values` qui retourne une séquence dont les dou-

blons auront été retirés, `avg` qui retourne la moyenne d'une séquence d'entiers,.... Elle autorise aussi l'imbrication de requêtes, à chaque endroit où une expression est utilisée, c'est-à-dire dans le `select` ou alors dans la partie droite du `from` ou encore dans une opération de comparaison dans une condition. Ainsi par exemple, nous pourrions avoir la requête imbriquée suivante, où la variable `x` qui est définie dans la requête du dessus est utilisée dans la requête du dessous.

```
select x + z
from x in [ 1 2 3 4],
     z in select y+1
          from y in [1 2 3]
          where x=y
```

L'un des derniers points importants est que l'on s'interdit l'utilisation des fonctions de ℂDuce dans les expressions, ou les types fonctions ($a \to b$) dans les motifs, ce qui nous retire la Turing-complétude. Cela nous permet d'offrir un langage déclaratif de manière à y apporter les optimisations que nous verrons dans le chapitre 4.

3.2.1.2 Le `select from where` avec expressions de chemins

Comme un des objectifs principaux de cette thèse est de pouvoir offrir les deux différents types de déconstructeurs que sont les expressions de chemins et le filtrage par motifs (voir section 2.2), nous offrons alors les projections à la XPath, en offrant ainsi une syntaxe proche de XQuery.

Nous définissons les projections comme étant une nouvelle expression ℂDuce, et qui correspondent aux expressions de chemins, dont la syntaxe est inspirée de XPath. Ces projections vont renvoyer des séquences d'éléments, et pourront être utilisées directement dans le `select from where`, notamment dans la clause `from`. Nous définissons trois types de projections qui sont la *projection sur élément*, la *projection sur attribut*, et la *projection sur descendants*.

Il est à noter que les motifs p_i d'une requête ℂQL sont vus ici comme de simples variables, comme l'autorise l'algèbre des motifs.

3.2.1.2.1 Projection sur élément

La projection sur un élément consiste à récupérer tous les éléments parmi les fils de type t d'un ensemble de nœuds. Sa syntaxe en ℂQL est la suivante :

$$e/t$$

où e représente une séquence d'éléments XML, et t un type. La projection de t renverra tous les fils de e qui ont ce type. Ce résultat sera une séquence ; ce qui implique, qu'on peut y appliquer une autre projection et ainsi les cumuler. L'expression $e/t_1/t_2$ renverra tous les petits-fils de e de type t_2, tels qu'ils aient un père de type t_1. L'expression XPath `//book` appliquée à un document XML respectant la DTD de la figure 2.3, sera équivalente en ℂQL à e/`<book ..>_`, où e représente la séquence singleton du document XML source. `<book ..>_` est un type ℂDuce représentant une valeur XML ayant pour balise `book`, et ayant possiblement des attributs[1], tout en ne s'intéressant pas à la descendance de cet élément. On peut aussi utiliser les déclarations de types ℂDuce de la figure 2.11 de la page 39, et écrire e/`Book`.

En XPath, la projection peut comporter des prédicats sur le chemin comme pour imposer que tel élément se trouve à telle position,...Ainsi, l'expression XPath : `//book[@year='1994']/title/text()` qui renvoie une séquence de nœuds texte pour les titres de chaque livre publié en 1994, pourra être écrite en ℂQL ainsi : e/`<book year="1994">_/<title ..>_/Char`. Cette dernière renverra une chaîne de caractères contenant tous les caractères de chacun des titres des livres publiés en 1994.

3.2.1.2.2 Projection sur attribut

La projection sur un attribut est similaire à la projection précédente. Sa syntaxe est :

$$\boxed{e/@a}$$

où e est une expression représentant une séquence d'éléments XML et a un atome ℂDuce (et non plus un type). Cette syntaxe est aussi proche de XPath. Par exemple, l'expression XPath `//book/@year` qui renvoie l'ensemble des attributs `year` des livres, sera équivalente à celle de ℂQL :e`/book/@year`.

3.2.1.2.3 Projection sur descendants

On dispose aussi de la projection sur les descendants qui sera écrite ainsi :

$$\boxed{e//t}$$

où e est une expression représentant une séquence d'éléments XML et t un type ℂDuce. Elle va renvoyer l'ensemble des descendants des éléments de e de type t, et ce, récursivement dans l'arbre. Si un élément a pour fils un élément de même type t alors ces deux éléments seront renvoyés dans une séquence.

[1] le `..` représente la possibilité d'attributs ; ne pas les mettre veut dire qu'il ne doit pas y avoir d'attributs.

3.2.2 Sémantique

Nous donnons dans cette partie la sémantique et les règles de typage du `select-from-where` et des différentes projections.

3.2.2.1 Le `select from where` avec motifs

La sémantique du `select-from-where` est définie sous forme opérationnelle par la traduction suivante dans ℂDuce :

```
transform e1 with p1 →
  transform e2 with p2 →
        ⋱
      transform en with pn →
        if c then [e0] else [ ]
```

L'opérateur `transform` de ℂDuce a été vu dans la section 2.3. Il joue le même rôle que la construction "`for`" de XQUERY Core [FSW00]. Cependant, il y a une différence particulière qui est que l'algorithme d'inférence de types et le schéma de compilation du filtrage par motifs est basé sur les automates d'arbres non-uniformes qui exploitent les types pour optimiser le code [Fri04a, Fri04b] ainsi que son exécution.

Dans la section 2.3, nous avons fait l'introduction de l'opérateur `transform` qui contient une liste de branches[2], sur lesquelles est appliquée une « *first match policy* ». Dans la traduction du `select-from-where`, qui est une cascade de `transform`, il n'y a qu'une seule branche appliquée par `transform`. Cette politique n'a donc plus de sens, puisque pour que l'opérateur `transform` soit évalué il faut que le filtrage de sa seule branche (et donc de son seul motif) réussisse. Ce qui implique que pour que la requête renvoie un résultat, tous les motifs doivent réussir pour un n-uplet v donné. Si il existe v tel que $v = (v_1, \ldots, v_n)$ et $v_i \in e_i$ (avec $i = 1 \ldots n$), et si pour chaque[3] v_i, $(v_i/p_i) \neq \Omega$ alors le résultat de la requête sera la substitution de e avec les variables capturées.

Cette traduction du `select from where` est donnée uniquement pour définir sa sémantique, elle n'est pas imposée à l'exécution. Nous pensons, en effet, que cette traduction doit être améliorée préalablement par un optimiseur de requêtes. Dans cette réécriture, la condition de la clause `where` est évaluée en dernier lieu. Par exemple, une optimisation simple pour un langage de requêtes en bases de données consiste à pousser les sélections directement là où elles portent, de manière

[2] Une branche est une expression de ce type : | $p_i \rightarrow e_i$

[3] Où Ω correspond à une erreur.

à réduire la taille des données intermédiaires. Ne pourrait-on pas évaluer une condition, si possible, dès la capture des sous-arbres ? Dans ce cas l'ajout de l'opérateur de test de condition (le `if-then-else`) entre des opérateurs `transform` pourrait être intéressant. Quelques optimisations dont celle-ci seront étudiées dans le chapitre 4.

3.2.2.1.1 Règle de typage de l'opérateur `transform`

Nous présentons dans la figure 3.1 la règle de typage de l'opérateur `transform`.

$$\frac{\begin{array}{c}(\text{for } s_i \equiv T\backslash(\lbag p_1 \rbag \mid \ldots \mid \lbag p_{i-1} \rbag) \wedge \lbag p_i \rbag) \\ \Gamma \vdash e : [T+] \quad \Gamma, (s_i/p_i) \vdash e_i : t_i\end{array}}{\Gamma \vdash \texttt{transform}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n : \bigvee_{\{i \mid s_i \not\simeq \mathbf{0}\}} t_i} \ (\textit{Transform})$$

FIG. 3.1 – Règle de typage de l'opérateur `transform`

Cette règle infère le type pour l'expression `transform` e `with` $p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n$. La notation $\lbag p_i \rbag$ correspond au calcul du type accepté par le motif p_i, et la notation (t_i/p_i) correspond à l'environnement de type calculé par le filtrage du motif p_i sur le type t_i : le type de chaque variable de p_i est ainsi inféré (ces notations sont définies dans [FCB02]). Pour chaque motif p_i, on calcule le type acceptant s_i en tenant compte de la police de priorité sur les motifs précédents. Ensuite on calcule le nouvel environnement (s_i/p_i) pour chaque p_i, ce qui va nous permettre de calculer le type t_i des expressions e_i. Le type inféré pour `transform` e `with` $p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n$ sera l'union de tous les t_i.

Le typage nous permet d'exprimer des contraintes assez fines. Par exemple, il est spécifié que l'expression e sur laquelle s'applique le `transform` est de type $[T+]$, et ne peut pas donc être vide. Nous pouvons alors rajouter cette règle dans le cas où e est une séquence vide.

$$\frac{\Gamma \vdash e : []}{\Gamma \vdash \texttt{transform}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n : []} \ (\textit{Transform2})$$

De même, nous pouvons lever une erreur, lorsque e n'a pas le type séquence, ce qui correspond à renvoyer la valeur[4] Ω.

$$\frac{\Gamma \vdash e \not\leq [\texttt{Any*}]}{\Gamma \vdash \texttt{transform}\ e\ \texttt{with}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n : \Omega} \ (\textit{Transform3})$$

3.2.2.1.2 Règle de typage du `select from where`

Comme l'itérateur `select from where` est constitué essentiellement de l'opérateur `transform` on peut déduire de la règle de typage de l'opérateur `transform` la règle de typage de la figure 3.2 pour le `select from where`.

[4] Qui est définie dans la sémantique de ℂDuce[FCB02].

$$\frac{\Gamma \vdash e_1 : [t_1+] \quad \Gamma,(t_1/p_1),\ldots,(t_{i-1}/p_{i-1}) \vdash e_i : [t_i+] \quad \Gamma,(t_1/p_1),\ldots,(t_n/p_n) \vdash e : t \quad \Gamma,(t_1/p_1),\ldots,(t_n/p_n) \vdash c : \mathit{Bool}}{\Gamma \vdash \texttt{select}\ e\ \texttt{from}\ p_1\ \texttt{in}\ e_1,\ \ldots,p_n\ \texttt{in}\ e_n\ \texttt{where}\ c\ :\ [\,t*\,]}$$

avec $var(p_i) \wedge var(p_j) = \varnothing$
$\forall i,j \in [1\ldots n],\ i \neq j$

FIG. 3.2 – Règle de typage pour les requêtes ℂQL

La condition c doit être de type `Bool`, et les e_i doivent être des séquences non vides possiblement hétérogènes[5] de type $[t_i+]$. Dans cette règle, (t/p) est l'environnement de typage produit par l'assignation des variables de p avec les meilleurs types[6] résultant du filtrage de p sur le type t. *var*(p) est l'ensemble des variables[7] se trouvant dans le motif p tel que défini ainsi dans [Fri04c] :

Définition 3.2.1 *var(p) Soit p un motif bien formé. Nous définissons l'ensemble des variables de p, noté var(p), par :*

var(p) $= \{x \ / x$ *accessible depuis p*$\} \cup \{x \ /\ (x := c)$ *accessible depuis* $p\}$

3.2.2.2 Le `select from where` avec expressions de chemins

Nous définissons les différentes projections par leur sémantique opérationnelle. Leurs règles de typage ne sont pas données quand elles sont composées uniquement de l'opérateur `transform`, il suffit d'appliquer celles définies dans la figure 3.1.

3.2.2.2.1 Projection sur élément La sémantique de e/t est définie sous forme opérationnelle :

```
transform e with
    <_ ..>[(x::t | _ )* ] → x
```

Il s'agit d'un opérateur `transform` appliqué à la séquence e, sur laquelle le motif `<_ ..>[(x::`t `| _ )* ]` est appliqué sur chacun de ses éléments. Ce motif capture dans `x` parmi les fils de chaque élément ceux qui sont de type t ; puis la séquence obtenue par la concaténation de tout ces `x` est renvoyée.

La projection e/t est équivalente à la requête utilisant le filtrage par motifs :

[5] Car le type t_i peut être une union de différents types (par exemple $[(s_1 \cup s_2 \cup s_3+)]$).

[6] Les définitions formelles et le calcul de la déduction des types meilleurs se trouvent dans [BCF03].

[7] La condition $var(p_i) \wedge var(p_j) = \varnothing$ pour $i \neq j$ n'est pas obligatoire mais est utile pour simplifier les preuves, et pour certains types d'optimisation.

```
flatten(select x
        from <_ ..>[(x::t|_)*] in e)
```

Comme x est une séquence, le `select` va alors renvoyer une séquence de cette séquence. Il faut alors l'applatir, d'où l'utilisation de l'opérateur `flatten`.

3.2.2.2.2 Projection sur attribut La sémantique de e/@a est définie sous forme opérationnelle :

```
transform e with
    <_ a=x>_ → x
```

La projection e/@a est équivalente à la requête utilisant le filtrage par motifs :

```
select x
from <_ a=x>_ in e
```

$$\frac{\Gamma \vdash e \ : \ [Any*] \qquad t \cap AnyXML \neq \varnothing}{\Gamma \vdash e//t \ : \ [t*]} \ (\textit{T'projDesc})$$

FIG. 3.3 – Règle de typage pour la projection sur descendants

3.2.2.2.3 Projection sur descendants La figure 3.3 correspond à la règle de typage pour l'opérateur de projection sur descendants. $e//t$ a pour type $[t*]$ si dans l'environnement de type Γ e est de type séquence d'arbres XML.

La projection sur descendants est du sucre syntaxique, dont la réécriture est donnée ci-dessus :

```
let xstack=ref [t*] []
in let f ( x : [Any*]) : [Any*] =
    xtransform x with
    xxx & t & <_ ..>[y::Any* ] → xstack := !xstack @ [xxx] ; f y
in let _ = f e in !xstack
```

Il n'est pas possible de capturer par un motif l'ensemble à n'importe quelle profondeur de sous-arbres de type t. Il existe en ℂDuce l'itérateur `xtransform` qui permet de capturer à n'importe quelle profondeur le premier sous-arbre filtrable par un motif, mais ne permet pas de continuer récursivement sur ce sous-arbre. Cela pose un problème en terme de typage, car on sort des langages réguliers

d'arbres[8]. Il faut donc passer par une pile inverse, qui utilise les références de ℂDuce, et utiliser une fonction récursive qui rappelle `xtransform` sur les fils des sous-arbres filtrables.

Au contraire des projections sur élément et attribut, il n'existe donc pas de requête ℂQL avec motifs équivalente.

3.2.3 Exemples de requêtes

Nous présentons le langage ℂQL par une série de requêtes qui illustrent ses déconstructeurs, son pouvoir expressif, et l'intérêt des motifs et de l'inférence de types de ℂDuce. Certaines de ces requêtes sont tirées des « *XML Query Use Cases* » [CFF+03], ou d'articles présentant ℂQL [BCM04, BCM05, CDu05].

3.2.3.1 Jointures

Les jointures et les produits cartésiens peuvent être facilement exprimés en ℂQL.

```
<books-with-prices>
select <book-with-prices>[t1
       <price-bstore2>([p2]/Char)
       <price-bstore1>([p1]/Char)]
from b in [doc]/Book ,
     t1 in [b]/Title,
     p1 in [b]/Price,
     e in [bstore2]/Entry,
     t2 in [e]/Title,
     p2 in [e]/Price
where t1=t2
```

```
type Reviews = <reviews>[ Entry + ]
type Entry = <entry>[ Title
                      Price
                      Review ]
type Title = <title>[ PCDATA ]
type Price = <price>[ Int ]
type Review = <review>[ PCDATA ]
```

FIG. 3.4 – Requête Q5 des « *XML Query Use Cases* »et le type de la valeur `bstore2`

3.2.3.1.1 Jointure La requête de la figure 3.4 donne pour chaque livre présent dans la bibliographie et dans `bstore2`, son titre et ses prix respectifs, le tout encapsulé dans une nouvelle balise `<book-with-prices>`. Il s'agit d'une jointure où pour chaque livre d'un document, on doit vérifier qu'il existe dans l'autre document. On a donc besoin de capturer les deux titres (`t1` et `t2`), et d'en faire une comparaison via la condition de jointure : `where t1 = t2`. On capture ensuite les deux prix (`p1` et `p2`), pour les encapsuler dans de nouvelles balises dans la clause

[8] Néanmoins des travaux sont en cours qui permettront d'approximer le type de ce genre d'opérateur [Ngu04].

`select` qui crée un nouvel élément `book` qui contiendra ainsi le titre, et les deux prix. Le résultat d'une requête étant une séquence, on peut l'encapsuler par une balise pour obtenir une valeur XML.

Cette requête utilise exclusivement les projections, on peut l'écrire (figure 3.5) plus concisément par des motifs.

```
<books-with-prices>
select <book-with-price>[t2 <price-bstore2>p2 <price-bstore1>p1]
from <bib>[b::Book*] in [bib],
     <book ..>[t1&Title _* <price>p1] in b,
     <reviews>[e::Entry*] in [bstore2],
     <entry>[t2&Title <price>p2;_] in e
where t1=t2
```

FIG. 3.5 – Requête Q5 des "*XML Query Use Cases*" utilisant le filtrage par motifs

3.2.3.2 Illustration de la puissance des motifs

Nous montrons ici la puissance de l'algèbre de motifs de ℂDuce. La requête suivante retourne tous les livres de `doc` mais en enlevant les éléments de type $Price$. L'élimination d'éléments est rendue possible par le motif de différence de types.

```
<bib>
select <book year=y> x
from <book year=y ..>[(x::Any\ Price)|_)* ] in [doc]/Book
```

On capture en `x` la séquence des sous-arbres de type $Book$ qui ne soit pas de type $Price$. Une telle requête en XQUERY serait écrite ainsi :

```
for $p in $bib/bibliography/paper
return <papier year="{($b/@year)}">
                  {$b/*[not(self::price)]}
       </papier>
```

En XQUERY, les éléments `<price>` sont retirés par l'utilisation du filtrage de XPATH, et de la fonction booléenne `not()`. À la différence du système de types de XQUERY, ℂDuce va inférer que le résultat n'aura jamais d'éléments `<price>`. Le type de la requête sera :

```
<bib>[ <book year=String>[ Title (Author+|Editor+) Publisher ]* ].
```

En XQUERY ce résultat serait :

```
<bib>[ <book year=String>[ Title (Author+|Editor+) Publisher ]*
Price?]
```

3.2.3.2.1 Attributs Nous pouvons exprimer aussi simplement des opérations de tests et de reconstruction sur les attributs. Par exemple, la requête suivante,

```
select <(b) (a\year)> x
from <(b) (a)>[ (x::(Any|_ )* ] in [doc]/Book
```

retire tous les attributs `year` (si ils existent) de tous les attributs éventuels des livres.

Nous pouvons aussi changer un certain attribut en un nouvel attribut, dans la requête suivante :

```
select <(b) (a\+annee=a.year)> x
from <(b) (a&year=_)>[(x::Any\Price|_ )*] in [doc]/Book
```

Cette requête permet de reconstruire un nouvel élément qui aura la même balise que le livre (capturée par `(b)`), et tel que l'attribut `year` (capturé par `(a)`) aura été renommé par `annee`, tout en retirant les éléments fils de type $Price$.

3.2.3.3 Précision de l'inférence de types

En ℂDuce la correction des types assure que les types déduits sont toujours un surtype des valeurs réellement calculées par ℂDuce. Étant donné un langage Turing-complet, la complétude de l'inférence de types ne peut pas être assurée. Cependant dans de nombreux cas, les résultats de l'inférence sont extrêmement précis grâce au sous-typage sémantique[9] de ℂDuce. Par exemple, le système de types peut calculer exactement le type différence $T \setminus U$, qui est le type contenant les valeurs qui sont dans T mais pas dans U, et l'intersection $T \cap U$ qui est le type contenant les valeurs présentes dans les T et U. Comme montré dans [FCB02], ce sont les opérations minimales pour inférer des types précis. Nous allons donner quelques exemples de requêtes ℂQL pour l'illustrer.

La requête suivante renvoie les livres ayant des éditeurs.

```
[bib]/<book ..>[_* Editor _*]
```

Pour cette requête, ℂDuce calcule le type précis suivant :

[9] Présenté à la page 38.

```
[ <book year=String>[ Title Editor+ Publisher Price ]* ]
```

qui est un type plus fin que le type $Book$ des types définis correspondant à la DTD de la figure 2.11 de la page 39. Cette précision est une particularité de ℂDuce. Par exemple, en XQUERY, la même requête formulée ainsi :

```
$bib/book[editor]
```

où `$bib` a pour type $Biblio$, aura pour type inféré :

```
[ <book year=String>[ Title (Author+ | Editor+ ) Publisher Price]* ]
```

qui ne considère pas le filtrage unique sur `editor`. Cela est dû au fait que le système de type de XQUERY est défini au dessus de XQUERY Core, qui est le langage minimal tel que toutes les expressions y sont traduites dès la compilation, avant l'opération de typage des expressions. Ce qui implique que la condition soit séparée de la sélection de l'axe `book`, et qu'elle soit traduite par un `if-then-else`. C'est pourquoi il y a perte de leur relation mutuelle lors de la compilation.

Pour second exemple considérons la requête de la figure 3.6 qui va extraire tous les livres qui n'ont pas un unique auteur. Cette requête garde tous les livres (`Any`) exceptés ceux ayant un seul auteur.

```
[doc]/(Any \ <book ..>[Title Author (Any\Author)*])
```

FIG. 3.6 – Quels sont les livres n'ayant pas un seul auteur ?

En retirant les livres n'ayant qu'un auteur, le système de types déduit qu'il y a soit au moins deux auteurs, soit aucun auteur (et donc uniquement des éditeurs). Le type inféré sera encore un type très précis :

```
[<book year=String>[ Title (Author Author+ |Editor +) Publisher Price]* ]
```

3.2.3.4 Utilisation des déconstructeurs

Nous illustrons la puissance expressive des requêtes par un exemple tiré des « *XML Query Use Cases* » [CFF+03]. Il s'agit de la requête Q1 : *Donner la liste des livres publiés par Addison-Wesley après l'année 1991, en incluant seulement l'année et le titre.* Pour cette requête il faut pouvoir extraire chaque livre, et pour chaque livre, leur titre, leur année et leur éditeur.

```
<bib>
select <book year=y> t
from <bib>[b::Book*] in [doc],
     <book year=y>[t::Title _+ <publisher>"Addison-Wesley" ;_] in b
where 1991 << int_of(y)
```

FIG. 3.7 – Requête Q1 utilisant le filtrage par motifs

On peut capturer tous ces éléments en deux motifs comme dans la figure 3.7. On extrait tous les livres dans la variable `b`, en un motif, puis pour chaque livre `b`, on extrait en un seul motif l'année dans la variable `y`, le titre dans la variable `t`, et on vérifie que la valeur capturée pour l'éditeur soit de type `<publisher>"Addison-Wesley"`. On peut noter aussi les utilisations du joker dans le dernier motif qui sera recompilé dans un motif équivalent : `<book year=y>[t::Title (Author+|Editor+) <publisher>"Addison-Wesley" Price]`. On peut noter l'utilisation de la fonction prédéfinie en ℂDuce, `int_of()`. Comme le type de `year` est le type *String* (`PCDATA` dans la DTD de la figure 2.11), contrairement à XQUERY, nous devons le convertir en entier pour pouvoir le comparer avec 1991.

```
<bib>
select <book year=y>([b]/Title)
from b in [doc]/Book,
     y in [b]/@year
where 1991<<int_of(y)
      and [b]/Publisher = [<publisher>"Addisson-Wesley"]
```

FIG. 3.8 – Requête Q1 utilisant la navigation par projection

Nous pourrions aussi utiliser les projections comme dans la figure 3.8 et non plus le filtrage précédent. Pour écrire une telle requête, nous avons besoin d'autant de projections que de sous-arbres à évaluer ou à renvoyer. Ce style de programmation reste très proche de la manière d'écrire la requête en XQUERY (voir figure 2.9 de la page 33). Le principal intérêt des projections, est de pouvoir écrire une requête en ne connaissant pas complètement la structure et surtout l'ordre des éléments d'une DTD. Si par exemple, l'ordre des éléments de type *Book* diffère, dans la figure 2.3 au lieu de

`<!ELEMENT book (title, (author+ | editor+ ), publisher, price )>`

nous avons

`<!ELEMENT book (price, title, (author+ | editor+ ), publisher )>`, le

filtrage de la requête de la figure 3.7 échouera, alors que la requête avec projection réussira.

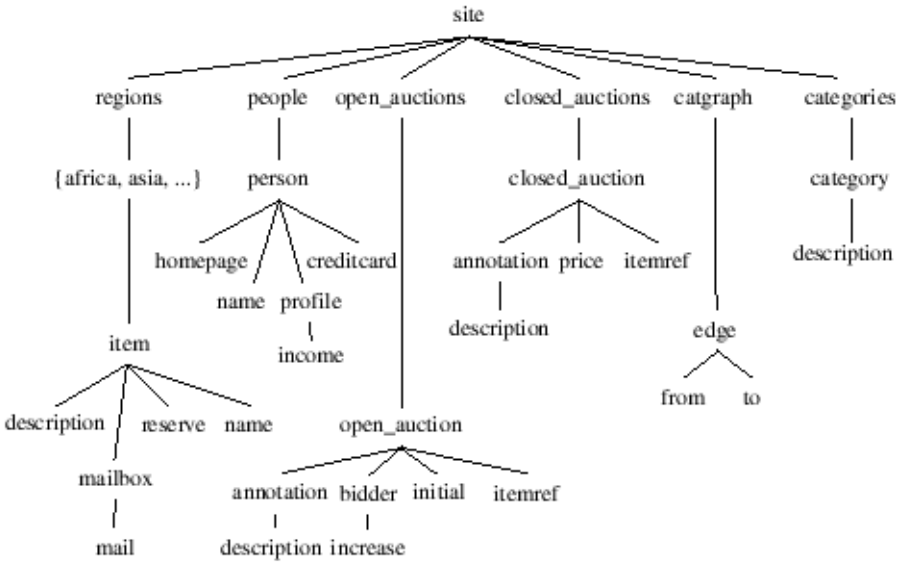


FIG. 3.9 – Relation entre les différents éléments d'un document XMark sous forme d'arbre

Les projections sont aussi très utiles aussi pour explorer en profondeur des documents profonds. Par exemple, dans XMark[SWK+02] (qui est un projet de Benchmarks pour XML), pour capturer l'ensemble des montants de toutes les enchères, en suivant le schéma de la figure 3.9 présenté dans [SWK+02], une seule projection suffit.

```
[document]/<open_auctions ..>_/<open_auction ..>_/<bidder ..>_/<increase ..>_
```

alors qu'avec un filtrage par motifs cela devient assez fastidieux :

```
select x4
from <site ..>[(x1::<open_auctions ..>_|_)*] in [document],
     <open_auctions ..>[(x2::<open_auction ..>_|_)*] in x1,
     <open_auction ..>[(x3::<bidder ..>_|_)*] in x2,
     <bidder ..>[(x4::<increase ..>_|_)*] in x3
```

Notons que l'on peut mélanger ces deux types de déconstructeurs des figures 3.7 et 3.8 pour écrire la requête de la figure 3.10. Cela permet de combiner à la fois la facilité d'écriture de la navigation verticale et le pouvoir expressif des motifs.

```
<bib>
select <book year=y> t
from <book year=y .. >[t::Title _+ <publisher>"Addison-Wesley";_] in [doc]/Book
where 1991 << int_of(y)
```

FIG. 3.10 – Requête Q1 utilisant la navigation par projection et le filtrage par motifs

On peut remarquer que XQUERY et ℂQL avec projections n'utilisent que des variables qui sont liées aux résultats de projections, alors que ℂQL avec motifs exploite pleinement l'algèbre de motifs de ℂDuce (voir figure 2.14 de la page 45).

Bien que ℂQL avec projections soit très proche et semble imiter XQUERY, il existe une importante différence entre ces deux langages outre la syntaxe. Ce dernier utilise une sémantique ensembliste sur les nœuds selon laquelle un même nœud ne peut apparaître plus d'une fois ; alors que le sémantique de ℂQL est multi-ensembliste et permet la copie de mêmes sous-arbres dans une séquence. La sémantique ensembliste peut être simulée en utilisant l'opérateur `distinct_values` sur chaque séquence.

Les projections sont aussi utiles pour capturer des éléments récursifs, ce que ne peut pas faire le filtrage par motifs. Considérons la DTD de la figure 3.11 inspirée de la partie TREE des « *XML Query Use Cases* ». Celle-ci représente le type d'un livre ayant des sections imbriquées entre elles à différentes profondeurs :

```
<!ELEMENT book (title, section+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT section (title, (p | section)* )>
<!ATTLIST section id ID #IMPLIED
<!ELEMENT p (#PCDATA)>
```

FIG. 3.11 – DTD avec entités récursives

Pour capturer l'ensemble des attributs `id` de chaque section à partir du document de ce type `book` et représenté par la variable `livre`, une projection avec descendants suffit :

```
[livre]//section/@id
```

Si on ne connaît pas le document, et qu'on ne dispose que de la DTD, il est alors impossible de savoir à quelle profondeur est le dernier élément `<section>`, pour pouvoir écrire un motif correspondant. Cette requête n'est donc pas exprimable avec des motifs, ni même avec des projections sur éléments, car aucun des deux ne permettent d'obtenir cette forme de fermeture transitive.

Le filtrage par motifs est également très agréable pour exprimer des contraintes de position. Considérons le motif suivant :

```
select [t]
from <book ..>[t&Title  _* <author>[<last>['B' _*] _]
                       <author>[<last>['S' _*] _] _*] in [doc]/Book
```

Cette requête renvoie l'ensemble des titres des livres tels qu'il y ait un auteur dont l'initiale du nom de famille soit la lettre B, et qui soit suivi immédiatement par un auteur dont l'initiale est S.

Elle s'exprimera ainsi en XPATH :

```
//book[author[starts-with(./last/text(),'S')
              and ./preceding-sibling::author
                      [position()=1
                       and starts-with(./last/text(),'B')]]]/title
```

3.3 Conclusion

Le langage de requêtes ℂQL a donc été défini dans ce chapitre avec une syntaxe déclarative et une sémantique opérationnelle, qui prennent en compte les deux types de déconstructeurs qui sont le filtrage par motifs, et la navigation par expressions de chemins. Ces deux types de déconstructeurs combinés permettent d'avoir des requêtes agréables à écrire comme celle présentée dans la figure 3.10.

Dans le chapitre suivant nous allons proposer des optimisations dont une qui permette de remplacer chaque projection par des motifs, permettant ainsi de capturer plusieurs éléments en un seul parcours de motif. Nous étudierons la préservation de la sémantique des requêtes pour une telle optimisation. Nous proposerons aussi une optimisation inspirée de la descente des sélections qui est une optimisation classique en bases de données

Ensuite, dans le chapitre d'après nous évaluerons ces différentes optimisations sur différentes approches de benchmarks qui nous permettront de valider ces optimisations.

Chapitre 4

Optimisations

Sommaire

4.1 Motivations 65

4.2 Traduction et désimbrication des projections en filtrage par motifs 66

4.2.1 Algorithme de traduction 66

4.2.2 Préservation de la sémantique 70

4.2.3 Préservation du typage 79

4.2.4 Exemple 82

4.3 Autres optimisations 83

4.3.1 Transformation en motifs d'une partie de la condition . . 84

4.3.2 Consolidation des motifs 85

4.3.3 Remontée des sélections 86

4.4 Conclusion 87

4.1 Motivations

Nous montrons dans cette section que toutes les projections utilisées dans une requête peuvent être réécrites dans une requête $\mathbb{C}$QL avec motifs. Si d'un point de vue théorique une telle traduction est intéressante, d'un point de vue pratique, elle ouvre la voie à de nombreuses optimisations ainsi qu'à un gain considérable de performances.

4.2 Traduction et désimbrication des projections en filtrage par motifs

L'utilisation du filtrage par motifs permet un gain de temps notable comparé aux expressions de chemins, du fait que les motifs peuvent capturer plusieurs éléments en une seule passe. Il est assez évident que les projections[1] peuvent être encodées en motifs puisque leurs réécritures en motifs ont été montrées dans les sections précédentes. Nous nous intéressons donc à la réécriture des projections en motifs. Nous définissons formellement dans cette partie une traduction qui :

- va éliminer les projections en les réécrivant dans des motifs par l'algorithme de traduction $\mathcal{T}$.
- et va transformer si possible et de manière non exhaustive une série de conditions dans des motifs[2].

```
<bib>
select <book year=y>[t]
from b in [biblio]/Book ,
     p in [b]/Publisher ,
     t in [b]/Title ,
     y in [b]/@year
where (p = <publisher>"Addison-Wesley") and (int_of(y)>>1990)
```

FIG. 4.1 – Requête avec projections

D'un point de vue pratique, l'utilisation d'une projection dans une requête est équivalente à l'utilisation de requêtes imbriquées. Si nous réécrivons dans la requête de la figure 4.1 les requêtes associées aux projections, nous obtenons une requête avec un niveau d'imbrication (figure 4.2). L'objectif général d'une optimisation en bases de données est d'essayer de désimbriquer les requêtes ; nous proposons alors la traduction $\mathcal{T}$ qui va désimbriquer ces `select` comme on peut le voir dans la requête de la figure 4.8 à la page 83 qui correspond à la requête de la figure 4.1 après traduction.

4.2.1 Algorithme de traduction

Nous définissons un algorithme de réécriture de requêtes (dénotées par q) (non imbriquées) pouvant contenir des projections, en des requêtes (dénotées par $\bar{q}$)

[1] Excepté la projection sur descendants

[2] Par exemple la condition `where x=1` est équivalente au motif `x&1`

```
<bib>
select <book year=y>[t]
from b in (select v from <_ ..>[(yb::Book|_)*] in [biblio], v in yb)
     p in (select v from <_ ..>[(yp::Publisher|_)*] in [b], v in yp)
     t in (select v from <_ ..>[(yt::Title|_)*] in [b], v in yt)
     y in (select yy from <_ year=yy>_ in [b])
where (p = <publisher>"Addison-Wesley") and (int_of(y)>>1990)
```

FIG. 4.2 – Requête avec un niveau d'imbrication

Requêtes avec projections		Requêtes sans projection	
q	::= select e from f where c	$\bar{q}$	::= select $\bar{e}$ from $\bar{f}$ where $\bar{c}$
	| select e from f		| select $\bar{e}$ from $\bar{f}$
e	::= x	$\bar{e}$	::= x
	| v		| v
	| $[e \ldots e]$		| $[\bar{e} \ldots \bar{e}]$
	| flatten(e)		| flatten($\bar{e}$)
	| $e\langle\ \ell = e \ldots \ell = e \rangle\ e$		| $\bar{e}\langle\ \ell = \bar{e} \ldots \ell = \bar{e} \rangle\ \bar{e}$
	| (e,e)		| $(\bar{e},\bar{e})$
	| *op*(e)		| *op*($\bar{e}$)
	| e/t		
	| $e/@a$		
f	::= p in e, f	$\bar{f}$	::= p in $\bar{e}$, $\bar{f}$
	| p in e		| p in $\bar{e}$
c	::= ‘true	$\bar{c}$	::= ‘true
	| ‘false		| ‘false
	| not(c)		| not($\bar{c}$)
	| c or c		| $\bar{c}$ or $\bar{c}$
	| c and c		| $\bar{c}$ and $\bar{c}$
	| e *bop* e		| $\bar{e}$ *bop* $\bar{e}$
	| member(e,e)		| member($\bar{e}$,$\bar{e}$)

FIG. 4.3 – Grammaire des requêtes avec et sans projections

ne contenant plus de projections. q et $\bar{q}$ sont définis dans la figure 4.3, ainsi que e dénotant une expression pouvant contenir des projections, et son alternative $\bar{e}$ ne contenant pas de projection, f dénotant une séquence de motifs filtrés sur une

séquence d'expressions pouvant contenir des projections, et son alternative $\bar{f}$, ainsi c et $\bar{c}$ pour les conditions booléennes avec projections ou sans.

Le principe de la traduction va être de localiser les projections dans une requête, pour les supprimer et rajouter le motif adéquat. Pour cela, nous disposons des contextes d'évaluation. La grammaire de ces contextes est donnée dans la figure 4.4.

$$
\begin{array}{lll}
E[\ ] ::= & [\,] & \\
 & \mid\ [e_1 \ldots e_n\ E[\,]\ \bar{e}_1 \ldots \bar{e}_m] & m,n \geq 0 \\
 & \mid\ \texttt{flatten(}E[\,]\texttt{)} & \\
 & \mid\ \langle\ \bar{e}\ \ \ell = e \ldots \ell = e\ \rangle\ E[\,] & \\
 & \mid\ \langle\ \bar{e}\ \ \ell = e \ldots \ell = E[\,]\ \ell = \bar{e} \ldots \rangle\ \bar{e} & \\
 & \mid\ (E[\,], \bar{e}) & \\
 & \mid\ E[\,]/t & \\
 & \mid\ E[\,]/@a & \\
 & \mid\ \mathit{op}(E[\,]) & \\
 & & \\
C[\ ] ::= & [] & \\
 & \mid\ \texttt{not(}C[\,]\texttt{)} & \\
 & \mid\ c\ \texttt{or}\ C[\,] & \\
 & \mid\ C[\,]\ \texttt{or}\ \bar{c} & \\
 & \mid\ c\ \texttt{and}\ C[\,] & \\
 & \mid\ C[\,]\ \texttt{and}\ \bar{c} & \\
 & \mid\ e\ \mathit{bop}\ E[\,] & \\
 & \mid\ E[\,]\ \mathit{bop}\ \bar{e} & \\
 & \mid\ \texttt{member(}e\texttt{,}E[\,]\texttt{)} & \\
 & \mid\ \texttt{member(}E[\,]\texttt{,}\bar{e}\texttt{)} &
\end{array}
$$

FIG. 4.4 – Contextes d'évaluation

Les contextes $E[\,]$ et $C[\,]$ (qui correspondent respectivement aux expressions et aux conditions) suivent une stratégie « *innermost-rightmost*[3] » qui donne un ordre déterministe dans la recherche de projections. Par exemple, quand nous aurons à traduire dans la suite, l'expression de chemin $x/t_1/t_2$, la règle du contexte sur les expressions à considérer en premier est $[\,]/t_2$.

Nous définissons l'ensemble $bv(f)$ tel que :

Définition 4.2.1 *L'ensemble* $bv(f)$ *des variables liées est défini tel que :*
- $bv(p \text{ in } e,\ f) = \mathit{var}(p) \cup bv(f)$
- $bv(p \text{ in } e) = \mathit{var}(p)$

[3]Le plus profond puis le plus à droite d'abord

Comme nous devons créer de nouveaux motifs pour remplacer les projections, cet ensemble va nous permettre de créer de nouvelles variables libres, en regardant si elles n'appartiennent pas à cet ensemble qui contiendra toutes les variables liées.

Nous donnons l'algorithme $\mathcal{T}$ de réécriture qui prend une requête q correspondant à la grammaire de la figure 4.3, et qui renvoie une requête $\bar{q}$ tel que les projections de q aient été remplacées, et que ces deux requêtes aient la même sémantique.

Définition 4.2.2 *Soient les expressions et les contextes d'évaluation définis dans la figure 4.4. La traduction $\mathcal{T}$ est découpée en trois étapes successives. La première (S) déplace les projections de la clause* `select` *dans la clause* `from`*, la seconde (W), de la clause* `where` *vers la clause* `from`*, et la dernière (F) élimine les projections en les remplaçant par des motifs* ad hoc.

S1 : `select` $E[\bar{e}/t]$ `from` f `where` $c \rightarrow_{\mathcal{S}}$ `select` $E[x]$ `from` f , x `in` `[`$\bar{e}/t$`]` `where` c, $x \notin bv(f)$

S2 : `select` $E[\bar{e}/@a]$ `from` f `where` $c \rightarrow_{\mathcal{S}}$ `select` $E[x]$ `from` f , x `in` `[`$\bar{e}/@a$`]` `where` c, $x \notin bv(f)$

W1 : `select` $\bar{e}_0$ `from` f `where` $C[\bar{e}/t] \rightarrow_{\mathcal{W}}$ `select` $\bar{e}_0$ `from` f , x `in` `[`$\bar{e}/t$`]` `where` $C[x]$, $x \notin bv(f)$

W2 : `select` $\bar{e}_0$ `from` f `where` $C[\bar{e}/@a] \rightarrow_{\mathcal{W}}$ `select` $\bar{e}_0$ `from` f , x `in` `[`$\bar{e}/@a$`]` `where` $C[[x]]$, $x \notin bv(f)$

F1 : `select` $\bar{e}_0$ `from` $\bar{f}$, p `in` $E[\bar{e}/t]$, f `where` $\bar{e} \rightarrow_{\mathcal{F}}$

- *Si le type de* $\bar{e}$ *est un sous-type de* `[Any]` *alors* `select` $\bar{e}_0$ `from` $\bar{f}$, `<_ ..>[(`x `::t|_)*]` `in` $\bar{e}$, p `in` $E[x]$, f `where` $\bar{e}$, $x \notin bv(\bar{f}) \cup bv(f)$
- *sinon* `select` $\bar{e}_0$ `from` $\bar{f}$, `[(<_ ..>[(`x `::t|_)*] | `x `::'nil)*]` `in` `[`$\bar{e}$`]` , p `in` E`[flatten(`x`)]` , f `where` $\bar{e}$, $x \notin bv(\bar{f}) \cup bv(f)$

F2 : `select` $\bar{e}_0$ `from` $\bar{f}$, p `in` $E[\bar{e}/@a]$, f `where` $\bar{e} \rightarrow_{\mathcal{F}}$

- *Si le type de* $\bar{e}$ *est un sous-type de* `[Any]` *alors* `select` $\bar{e}_0$ `from` $\bar{f}$, `<_` a`=`x`>_` `in` $\bar{e}$, p `in` $E[[x]]$, f `where` $\bar{e}$, $x \notin bv(\bar{f}) \cup bv(f)$
- *sinon* `select` $\bar{e}_0$ `from` $\bar{f}$, `[(<_` a`=`x`>_ | `x `::'nil)*]` `in` `[`$\bar{e}$`]` , p `in` $E[[x]]$, f `where` $\bar{e}$, $x \notin bv(\bar{f}) \cup bv(f)$

Nous avons défini les règles de réécriture S1, S2, W1, W2, F1 et F2 de manière qu'elles soient appliquées uniquement l'une après l'autre, et autant de fois que nécessaire.

Ainsi après application d'un certain nombre de fois S1, on a la garantie qu'il n'y a plus de projections sur élément dans la clause `select`. Après S2, on n'aura plus

de projections sur attributs. Et le contexte $E[\,]$ nous donne un ordre sur les projections à considérer.
Les règles W1, W2, F1, F2 ne pourront pas s'appliquer tant que l'application de toutes les règles S1 et S2 n'aura pas terminé, car ces premières demandent une expression $\bar{e}$ dans la clause `select`.
Une fois que toutes les règles S1 et S2 auront terminé, alors les règles W1 et W2 s'appliqueront tant qu'il reste des projections dans la clause `where`, les règles F1 et F2 demandant une expression $\bar{c}$ dans la clause `where`.
Et dans les règles F1 et F2, les premières lignes sont considérées avant les suivantes. La condition $x \notin bv(f)$ nous permet d'introduire des variables fraîches qui éviteront d'utiliser des variables déjà définies dans la clause `from`.

On se place dans le cas de requêtes non-imbriquées, mais cette réécriture peut s'étendre facilement en cas de requêtes imbriquées, en considérant la traduction de chaque requête une à une, et en utilisant, par exemple, une stratégie « *innermost-rightmost* ».

On notera $\lfloor q \rfloor_{\mathcal{T}}$ le résultat de la traduction parpour poruver le $\mathcal{T}$ sur la requête q. Notons que puisqu'une seule règle ne peut s'appliquer à la fois, et qu'il n'y a donc pas de paires critiques, alors $\lfloor q \rfloor_{\mathcal{T}}$ est une fonction.
$\lfloor q \rfloor_{\mathcal{T}}$ est également une fonction totale, puisque il y a terminaison. En effet, les règles S1 et S2 diminuent, à chaque pas, le nombre de projections de la clause `select` en les déplaçant vers la clause `from`, les règles W1 et W2 diminuent, à chaque pas, le nombre de projections de la clause `where` en les déplaçant vers la clause `from`, et les règles F1 et F2 diminuent, à chaque pas, le nombre de projections de la clause `from`.

Nous allons montrer que la sémantique puis que le typage sont préservés par cette traduction.

4.2.2 Préservation de la sémantique

Pour prouver que cette traduction préserve la sémantique, nous avons besoin de définir la réduction d'une requête en une valeur $\mathbb{C}$Duce. Nous considérons la réduction $\rightarrow$ définie dans la figure 4.6 avec une sémantique à petits pas inspirée de celle de[CF05] ; et nous définissons aussi dans la figure 4.5 les valeurs et les expressions $\mathbb{C}$Duce, dans laquelle nous rajoutons les opérateurs `transform`, `if then else` et `@` (pour la concaténation). Nous avons besoin de définir la réduction de ces opérateurs car ils sont les composants principaux d'une requête.

Nous avons aussi besoin de définir un contexte d'évaluation $R[\,]$, qui nous

$$
\begin{array}{lcl}
v & ::= & \texttt{`true} \mid \texttt{`false} \mid \texttt{`nil} \mid (v,v) \mid c \\
e & ::= & x \mid (e,e) \mid e\texttt{@}e \mid \texttt{flatten}(e) \\
 & \mid & \texttt{transform } e \texttt{ with } p \texttt{ -> } e \\
 & \mid & \texttt{if } e \texttt{ then } e \texttt{ else } e
\end{array}
$$

FIG. 4.5 – Définition du langage cible

$$
\begin{array}{lcl}
\texttt{transform } (v,e_1) \texttt{ with } p \texttt{ -> } e_2 & \rightarrow & e_2[v/p]\texttt{@ transform } e_1 \texttt{ with } p\texttt{-> } e_2 \\
\texttt{transform `nil with } p \texttt{ -> } e_2 & \rightarrow & \texttt{`nil} \\
\texttt{if `true then } e_1 \texttt{ else } e_2 & \rightarrow & e_1 \\
\texttt{if `false then } e_1 \texttt{ else } e_2 & \rightarrow & e_2 \\
(e_1,e_2)\texttt{@}e_3 & \rightarrow & (e_1,e_2\texttt{@}e_3) \\
\texttt{`nil@}e & \rightarrow & e \\
\texttt{flatten}((e_1,e_2),e_3) & \rightarrow & (e_1,\texttt{flatten}(e_2,e_3)) \\
\texttt{flatten(`nil)} & \rightarrow & \texttt{`nil}
\end{array}
$$

FIG. 4.6 – Définition de la réduction $\rightarrow$

permette de réduire les expressions en valeurs v qui peuvent contenir des variables, et qui est décrit dans la figure 4.7. Les séquences et les valeurs XML peuvent être encodées par des paires, et c correspond aux constantes $\mathbb{C}$Duce. Nous ne considérons pas les fonctions, ni les références ce qui fait que ce langage cible ne possède pas d'effet de bord.

Ce qui nous permet d'obtenir la propriété suivante sur le contexte $R[\,]$:

$$\frac{e \rightarrow e'}{R[e] \rightarrow R[e']}$$

$$
\begin{array}{lcl}
R[\,] & ::= & (R[\,],e) \mid (e,R[\,]) \mid R[\,]\texttt{@}e \mid e\texttt{@}R[\,] \mid \texttt{flatten}(R[\,]) \\
 & \mid & \texttt{transform } R[\,] \texttt{ with } p \texttt{ -> } e \\
 & \mid & \texttt{transform } e \texttt{ with } p \texttt{ -> } R[\,] \\
 & \mid & \texttt{if } R[\,] \texttt{ then } e \texttt{ else } e \\
 & \mid & \texttt{if } e \texttt{ then } R[\,] \texttt{ else } e \\
 & \mid & \texttt{if } e \texttt{ then } e \texttt{ else } R[\,] \\
 & \mid & [\,]
\end{array}
$$

FIG. 4.7 – Définition du contexte $R[]$

Il est à noter que dorénavant, nous utiliserons de façon implicite la propriété

de confluence de $\mathbb{C}$Duce, et en particulier que si $e \rightarrow^* v$ et si $e \rightarrow^* v'$ alors $v = v'$.

Nous disposons maintenant du langage cible, qui va nous permettre de prouver dans la suite que les deux expressions d'une règle de réécriture de $\mathcal{T}$ se réduisent par $\rightarrow$ en une même expression e.

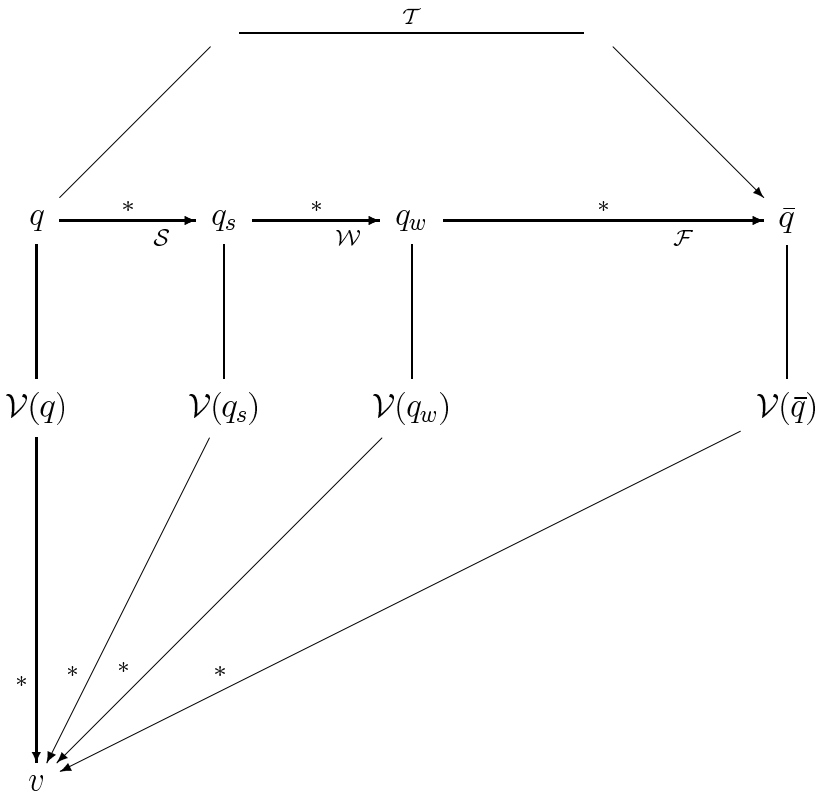
Définition 4.2.3 *Nous définissons la fonction $\mathcal{V}()$ qui associe à une requête sa sémantique. La définition est la suivante :*

$\mathcal{V}($`select` e `from` p_1 `in` e_1`, ...,` p_n `in` e_n `where` $c$$)$ $=$ `transform` e_1 `with` p_1 `->` `...transform` e_n `with` p_n`->` `if` c `then [e] else []`

Nous allons prouver la préservation de la sémantique de $\mathcal{T}$. Comme nous disposons d'un ordre de réduction sur $\mathcal{T}$ (les règles S1 et S2 sont toutes évaluées en premier, puis les règles W1 et W2, puis les règles F1 et F2), nous définissons trois nouvelles réductions $\rightarrow_{\mathcal{S}}$, $\rightarrow_{\mathcal{W}}$ et $\rightarrow_{\mathcal{F}}$, qui vont décrire respectivement les applications des règles S1 S2 , W1 W2 et F1 F2, de telle sorte que : si $\lfloor q \rfloor_{\mathcal{T}} = \bar{q}$ alors $q \rightarrow_{\mathcal{S}}^* q_s \rightarrow_{\mathcal{W}}^* q_w \rightarrow_{\mathcal{F}}^* \bar{q}$.

Nous allons nous servir de ce découpage pour prouver la relation suivante : si $\mathcal{V}(q) \rightarrow^* v$ et $\lfloor q \rfloor_{\mathcal{T}} = \bar{q}$ alors $\mathcal{V}(q_s) \rightarrow^* v$, $\mathcal{V}(q_w) \rightarrow^* v$ et $\mathcal{V}(\bar{q}) \rightarrow^* v$

Cette relation est illustrée dans la figure suivante :



La traduction $\mathcal{T}$ a une sémantique à grand pas et va réduire la requête q en $\bar{q}$, et également avec un certain nombre d'étapes de réduction $\rightarrow_{\mathcal{S}}$,$\rightarrow_{\mathcal{W}}$ et $\rightarrow_{\mathcal{F}}$ en

la même requête $\bar{q}$. Cette requête sans projection considérée avec sa sémantique opérationnelle va se réduire par $\rightarrow$ (définie dans la figure 4.6) dans la même valeur v que la réduction de $\mathcal{V}(q)$.

Nous allons donc montrer que chaque réduction $\rightarrow_{\mathcal{S}}$, $\rightarrow_{\mathcal{W}}$ et $\rightarrow_{\mathcal{F}}$ préserve la sémantique, et qu'ainsi $\mathcal{V}(q)$ se réduira en une valeur v, si $\mathcal{V}(\lfloor q \rfloor_{\mathcal{T}})$ se réduit en cette même valeur v.

Lemme 4.2.4 (Préservation de la sémantique pour $\rightarrow_{\mathcal{S}}$) *: Si* $\mathcal{V}(q) \rightarrow^* v$ *et si* $q \rightarrow_{\mathcal{S}}^* q_s$ *alors* $\mathcal{V}(q_s) \rightarrow^* v$.

Preuve :
Par induction sur la longueur n de la réduction $q \rightarrow_{\mathcal{S}}^n q_s$.

Cas de base :
$n = 0$. Il n'y a pas d'application des règles S1 et S2. On est dans le cas où il n'y a pas de projection. Alors $q_s = q$ et donc $\mathcal{V}(q_s) \rightarrow^* v$.

Cas inductif :
Prouvons la préservation de la sémantique pour chaque cas S1 et S2 des règles de réécriture de la fonction totale $\rightarrow_{\mathcal{S}}$. On suppose que $\mathcal{V}(q) \rightarrow^* v$, et $q \rightarrow_{\mathcal{S}} q' \rightarrow_{\mathcal{S}}^n q_s$, on va montrer que $\mathcal{V}(q') \rightarrow^* v$, d'où le résultat suit par hypothèse d'induction.
Montrons que la règle de réduction S1 préserve la sémantique.
L'expression $\mathcal{V}(q)$ est égale à
`transform` e_1 `with` p_1 `->`
...
`transform` e_m `with` p_m `->`
`if` c `then` $[E[\bar{e}/t]]$
`else []`
Considérons l'application du contexte $R[\,]$ sur $\mathcal{V}(q)$, et e'_1 tels que
$\mathcal{V}(q) = R[$`if` c `then` $[E[\bar{e}/t]]] = R[e'_1]$
ainsi $R[e'_1] \rightarrow^* v$
Considérons maintenant q' tel que $\mathcal{V}(q')$ est égal à
`transform` e_1 `with` p_1 `->`
...
`transform` e_m `with` p_m `->`
`transform` $[\bar{e}/t]$ `with` x`->`
`if` c `then` $[E[x]]$
`else []`

Soit e'_2 tel que
$\mathcal{V}(q')$=R[`transform [`$\bar{e}$`/`t`] with` x`->if` c `then [`$E[x]$`] else [] ]`=$R[e'_2]$
Ce choix d'expression e'_2 sur le contexte $R[\,]$ sur $\mathcal{V}(q')$ n'est pas anodin, car il permet de disposer du même contexte que pour $\mathcal{V}(q)$. Ainsi si on montre que ces deux expressions e'_1 et e'_2 se réduisent en une même valeur, alors on prouvera que $\mathcal{V}(q')\rightarrow^* v$.
Comme les séquences sont du sucre syntaxique de paires,
$R[e'_2]$=R[`transform (`$\bar{e}$`,'nil) with` x`->if` c `then [`$E[x]$`] else [] `](4)
En appliquant les règles de réduction de la figure 4.6, on obtient que
$R[e'_2]\rightarrow R$[`(if` c `then [`$E[x]$`] else [])`$[(\bar{e}/t)/x]$]
Puisque x est une variable fraîche, ce qui implique qu'elle n'apparaît que dans $E[x]$, on en déduit que R[`if` c `then [`$E[x]$`] else [] `$[(\bar{e}/t)/x]$] $\rightarrow$ R[`if` c `then [`$E[\bar{e}/t]$`] else []`]
On a donc ici que $R[e'_2]\rightarrow^* R[e'_1]$. Et donc que $\mathcal{V}(q')\rightarrow^*\mathcal{V}(q)$. Et finalement par transitivité que $\mathcal{V}(q')\rightarrow^* v$.
On peut donc en déduire que la sémantique est préservée pour la règle S1.
Le principe est le même pour la règle S2 :

```
transform e1 with p1 ->                transform e1 with p1 ->
...                                    ...
transform em with pm ->     →S         transform em with pm ->
if c  then [E[ē/@a]]                   transform [ē/@a] with x->
      else []                          if c  then [E[x]]
                                             else []
```

Et de façon identique à S1, on prouve que la sémantique est préservée pour S2.
On a alors bien que si $\mathcal{V}(q)\rightarrow^* v$ et si $q\rightarrow_{\mathcal{S}} q'\rightarrow_{\mathcal{S}}{}^n q_s$ alors par hypothèse d'induction $\mathcal{V}(q_s)\rightarrow^* v$

□

Lemme 4.2.5 (Préservation de la sémantique pour $\rightarrow_{\mathcal{W}}$) *: Si* $\mathcal{V}(q)\rightarrow^* v$ *et si* $q\rightarrow_{\mathcal{W}}{}^* q_w$ *alors* $\mathcal{V}(q_w)\rightarrow^* v$

Preuve :
Identique au lemme précédent. □

Lemme 4.2.6 (Préservation de la sémantique pour $\rightarrow_{\mathcal{F}}$) *: Si* $\mathcal{V}(q)\rightarrow^* v$ *et si* $q\rightarrow_{\mathcal{F}}{}^* \bar{q}$ *alors* $\mathcal{V}(\bar{q})\rightarrow^* v$

Preuve :

Par induction sur la longueur n de la réduction $q \rightarrow_{\mathcal{F}}^{n} \bar{q}$.

Cas de base :

$n = 0$. Il n'y a pas d'application des règles F1 et F2. On est dans le cas où il n'y a pas de projection. Alors $\bar{q} = q$ et donc $\mathcal{V}(\bar{q}) \rightarrow^{*} v$.

Cas inductif :

Prouvons par induction la préservation de la sémantique par chaque cas des règles de réécriture de la fonction totale $\rightarrow_{\mathcal{S}}$. On suppose que $\mathcal{V}(q) \rightarrow^{*} v$, et $q \rightarrow_{\mathcal{F}} q' \rightarrow_{\mathcal{F}} \bar{q}$, on va montrer que $\mathcal{V}(q') \rightarrow^{*} v$, d'où le résultat suit par hypothèse d'induction.

cas F1

F1 : `select` $\bar{e}_0$ `from` $\bar{f}$, p `in` $E[\bar{e}/t]$, f `where` $\bar{e} \rightarrow_{\mathcal{F}}$ Si le type de $\bar{e}$ est un sous-type de `[Any]` alors `select` $\bar{e}_0$ `from` $\bar{f}$, `<_ ..>[(`x `::`t`|_)*]` `in` $\bar{e}$, p `in` $E[x]$, f `where` $\bar{e}$, $x \notin bv(\bar{f}) \cup bv(f)$ sinon `select` $\bar{e}_0$ `from` $\bar{f}$, `[(<_ ..>[(`x `::`t`|_)*] |` x `::'nil)*]` `in` `[`$\bar{e}$`]` , p `in` E`[flatten(`x`)]` , f `where` $\bar{e}$, $x \notin bv(\bar{f}) \cup bv(f)$

On a donc $\mathcal{V}(q) =$

```
transform e1 with p1 ->
...
transform E[ē/t] with pi ->
...
transform em with pm ->
if c̄ then [ēm+1] else []
```

Soient $R[\,]$ et e'_1 tels que $\mathcal{V}(q) = R$

```
[ transform E[ē/t] with pi ->
  ...
  transform em with pm ->
  if c̄ then [ēm+1] else [] ]
```

$= R[e'_1]$

qui est équivalent à :

$R[e'_1] = R$

```
[ transform E[transform ē with <_ ..>[(y :: t|_)*]->y] with pi ->
  ...
  transform em with pm ->
  if c̄ then [ēm+1] else [] ]
```

où y est une variable fraîche.

Pour pouvoir réduire l'opérateur dans le contexte $E[\,]$, nous avons besoin de réduire $\bar{e}$. Soit $\bar{e} \rightarrow^{*} v$. Pour que la requête soit bien typée il faut que

$\bar{e}$ soit une séquence. On a besoin d'étudier le cas où v est une séquence d'une seule valeur ou contenant une séquence de plusieurs valeurs.

Supposons que le type de $\bar{e}$ est un sous-type de `[Any]`.

Soit $\bar{e} \rightarrow^* (v_1, \text{`nil})$:

On a donc que

$$R[e'_1] \rightarrow R \begin{bmatrix} \texttt{transform } E[y[v_1/\texttt{<_ ..>[(}y :: t\texttt{|_)*]]] with } p_i \texttt{ ->} \\ \ldots \\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->} \\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else []} \end{bmatrix}$$

Montrons maintenant que $\mathcal{V}(q')$ se réduit en cette même expression.

On a que $\mathcal{V}(q')=$

```
transform e1 with p1 ->
...
transform ē with <_ ..>[(x :: t|_)*] ->
transform E[x] with pi ->
...
transform em with pm ->
if c̄ then [ēm+1] else []
```

Soient $R[\,]$ comme défini plus haut et e'_2 tels que :

$$\mathcal{V}(q') = R \begin{bmatrix} \texttt{transform } \bar{e} \texttt{ with <_ ..>[(}x :: t\texttt{|_)*] ->} \\ \texttt{transform } E[x] \texttt{ with } p_i \texttt{ ->} \\ \ldots \\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->} \\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else []} \end{bmatrix} = R[e'_2]$$

et d'après la supposition que $\bar{e} \rightarrow^* (v_1, \text{`nil})$, on a donc que :

$$R[e'_2] \rightarrow R \begin{bmatrix} \texttt{transform } (v_1\texttt{,`nil}) \texttt{ with <_ ..>[(}x :: t\texttt{|_)*] ->} \\ \texttt{transform } E[x] \texttt{ with } p_i \texttt{ ->} \\ \ldots \\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->} \\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else []} \end{bmatrix}$$

et donc que l'expression précédente se réduit en :

$$R \begin{bmatrix} \texttt{(transform } E[x] \texttt{ with } p_i \texttt{ ->} \\ \ldots \\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->} \\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else [])}[v_1/\texttt{<_ ..>[(}x :: t\texttt{|_)*]} \end{bmatrix}$$

Comme la seule variable qui apparaît dans le motif `<_ ..>[(`$x :: t$`|_)*]` est x, et est une variable qui n'apparaît que dans l'expression $E[x]$, alors on peut déplacer la substitution $[v_1/\texttt{<_ ..>[(}x :: t\texttt{|_)*]}]$ directement sur

$E[x]$.

Ce qui fait que l'expression précédente se réduit (à une α-équivalence près) en :

$$R\left[\begin{array}{l}\texttt{transform } E[y[v_1/\texttt{<_ ..>[(}y :: t\texttt{|_)*]}]] \texttt{ with } p_i \texttt{ ->} \\ \ldots \\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->} \\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else []}\end{array}\right]$$

On a donc que l'expression $R[e'_2]$ s'est réduite en une même expression que $R[e'_1]$, donc que si $\mathcal{V}(q)\rightarrow^* v$ alors $\mathcal{V}(q'')\rightarrow^* v$, et donc par hypothèse d'induction que $\mathcal{V}(q)\rightarrow^*$ et si $q\rightarrow_{\mathcal{F}}{}^*\bar{q}$ alors $\mathcal{V}(\bar{q})\rightarrow^* v$

On a donc prouvé que dans le cas où le type de $\bar{e}$ est un sous-type de `[Any]` la sémantique est préservée.

Étudions maintenant le second cas, où le type de $\bar{e}$ est un sous-type de `[Any Any+]`, c'est-à-dire que $\bar{e}$ ne peut se réduire qu'en une valeur $(v_1,(v_2,(\ldots,(v_n,\texttt{`nil})\ldots)))$.

On a supposé que :

$$R[e'_1]{=}R\left[\begin{array}{l}\texttt{transform } E[\texttt{transform } \bar{e} \texttt{ with <_ ..>[(}y :: t\texttt{|_)*]->}y] \texttt{ with } p_i \texttt{ ->} \\ \ldots \\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->} \\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else []}\end{array}\right]$$

et que le type de $\bar{e}$ est un sous-type de `[Any Any+]`

Soit $\bar{e}\rightarrow^* (v_1,(v_2,(\ldots,(v_n,\texttt{`nil})\ldots)))$

On a donc que :

$R[e'_1]\rightarrow^*$

$$R\left[\begin{array}{l}\texttt{transform} \\ \quad E[\texttt{transform } (v_1,(v_2,(\ldots,(v_n,\texttt{`nil})\ldots))) \texttt{ with <_ ..>[(}y :: t\texttt{|_)*]->}y] \\ \texttt{with } p_i \texttt{ ->} \\ \ldots \\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->} \\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else []}\end{array}\right]$$

Cette expression se réduit en :

$$R\left[\begin{array}{l}\texttt{transform} \\ E[y[v_1/\texttt{<_ ..>[(}y :: t\texttt{|_)*]}] \texttt{ @}\ldots\texttt{@}y[v_n/\texttt{<_ ..>[(}y :: t\texttt{|_)*]}]] \texttt{ with } p_i \texttt{ ->} \\ \ldots \\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->} \\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else []}\end{array}\right]$$

Montrons maintenant que $\mathcal{V}(q')$ se réduit en cette même expression.

On a que $\mathcal{V}(q')=$

transform e_1 with p_1 ->

...

transform [$\bar{e}$] with [(<_ ..>[($x :: t$|_)*] | x ::'nil)*]] ->

transform E[flatten(x)] with p_i ->

...

transform e_m with p_m ->

if $\bar{c}$ then [$\bar{e}_{m+1}$] else []

Soient $R[\,]$ comme défini plus haut et e'_2 tels que :

$$\mathcal{V}(q')=R\begin{bmatrix}\texttt{transform [}\bar{e}\texttt{] with [(<_ ..>[(}x :: t\texttt{|_)*] | }x\texttt{::'nil)*]] ->}\\ \texttt{transform }E\texttt{[flatten(}x\texttt{)] with }p_i\texttt{ ->}\\ \texttt{...}\\ \texttt{transform }e_m\texttt{ with }p_m\texttt{ ->}\\ \texttt{if }\bar{c}\texttt{ then [}\bar{e}_{m+1}\texttt{] else []}\end{bmatrix}=R[e'_2]$$

et d'après la supposition que $\bar{e}\rightarrow^*$ (v_1,(v_2,(...,(v_n,'nil)...))), on a donc que :

$R[e'_2]\rightarrow^*$

$$R\begin{bmatrix}\texttt{transform ((}v_1\texttt{,(}v_2\texttt{,(...,(}v_n\texttt{,'nil)...))),'nil)}\\ \texttt{with [(<_ ..>[(}x :: t\texttt{|_)*] | }x\texttt{::'nil)*]] ->}\\ \texttt{transform }E\texttt{[flatten(}x\texttt{)] with }p_i\texttt{ ->}\\ \texttt{...}\\ \texttt{transform }e_m\texttt{ with }p_m\texttt{ ->}\\ \texttt{if }\bar{c}\texttt{ then [}\bar{e}_{m+1}\texttt{] else []}\end{bmatrix}$$

Cette expression se réduit en :

$$R\begin{bmatrix}\texttt{(transform }E\texttt{[flatten(}x\texttt{)] with }p_i\texttt{ ->}\\ \texttt{...}\\ \texttt{transform }e_m\texttt{ with }p_m\texttt{ ->}\\ \texttt{if }\bar{c}\texttt{ then [}\bar{e}_{m+1}\texttt{] else [])}\\ \texttt{[(}v_1\texttt{,(}v_2\texttt{,(...,(}v_n\texttt{,'nil)...)))/[(<_ ..>[(}x :: t\texttt{|_)*] | }x\texttt{::'nil)*]]}\end{bmatrix}$$

Puisque la seule variable de ce motif est x, et qu'elle ne porte que sur E[flatten(x)], l'expression précédente est équivalente à :

$$R\begin{bmatrix}\texttt{transform}\\ E\texttt{[flatten(}x\texttt{[(}v_1\texttt{,(...,(}v_n\texttt{,'nil)...))/[(<_ ..>[(}x :: t\texttt{|_)*] | }x\texttt{::'nil)*]]}\\ \texttt{)] with }p_i\texttt{ ->}\\ \texttt{...}\\ \texttt{transform }e_m\texttt{ with }p_m\texttt{ ->}\\ \texttt{if }\bar{c}\texttt{ then [}\bar{e}_{m+1}\texttt{] else []}\end{bmatrix}$$

[(v_1,(v_2,(...,(v_n,'nil)...)))/[(<_ ..>[($x :: t$|_)*] | x ::'nil)*]] signifie qu'on applique le motif [(<_ ..>[($x :: t$|_)*]|x ::'nil)*] sur la séquence (v_1,(v_2,(...,(v_n,'nil)...))), ce qui revient à appliquer le

motif (<_ ..>[(x :: t|_)*] sur chaque élément de cette séquence. On a donc que :

$$R\left[\begin{array}{l}\texttt{transform}\\ E[\texttt{flatten([}x\texttt{][}v_1\texttt{/(<_ ..>[(}x\texttt{ :: }t\texttt{|_)*]]@...@}x\texttt{[}v_n\texttt{/(<_ ..>[(}x\texttt{ :: }t\texttt{|_)*]])]}\\ \texttt{with } p_i \texttt{ ->}\\ \ldots\\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->}\\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else []}\end{array}\right]$$

Et en appliquant la réduction sur le flatten :

$$R\left[\begin{array}{l}\texttt{transform}\\ E[x\texttt{[}v_1\texttt{/(<_ ..>[(}x\texttt{ :: }t\texttt{|_)*]]@...@}x\texttt{[}v_n\texttt{/(<_ ..>[(}x\texttt{ :: }t\texttt{|_)*]]}]\\ \texttt{with } p_i \texttt{ ->}\\ \ldots\\ \texttt{transform } e_m \texttt{ with } p_m \texttt{ ->}\\ \texttt{if } \bar{c} \texttt{ then [}\bar{e}_{m+1}\texttt{] else []}\end{array}\right]$$

Et à une α-réduction près on obtient la même expression que pour la réduction de $R[e'_1]$. On obtient alors que $R[e'_1]$ et $R[e'_2]$ se réduisent en une même valeur, et donc que si $\mathcal{V}(q)\rightarrow^* v$ alors $\mathcal{V}(q')\rightarrow^* v$.

On a donc prouvé que pour toutes les valeurs différentes de $\bar{e}$ (qui est forcément de type séquence par définition de la projection sur élément), l'application d'une étape F1 préserve la sémantique. Pour la règle F2, le principe est le même. Ainsi on a montré par induction que la sémantique est préservée par une analyse de cas pour la réduction $\rightarrow_{\mathcal{F}}$.

□

Théorème 4.2.7 (Préservation de la sémantique de $\mathcal{T}$) *:*

si $\mathcal{V}(q)\rightarrow^* v$ *alors* $\mathcal{V}(\lfloor q\rfloor_{\mathcal{T}})\rightarrow^* v$

Si pour la requête q, sa sémantique opérationnelle $\mathcal{V}(q)$ se réduit en une valeur v, et si la traduction $\mathcal{T}$ avec une sémantique à grand pas va réduire la requête q en un certain nombre d'étapes en une requête $\bar{q}$, alors l'expression $\mathcal{V}(\bar{q})$ se réduira en v.

Preuve :

Le théorème de la préservation de la sémantique de $\mathcal{T}$ est vérifié, par l'application successive des trois précédents lemmes. □

4.2.3 Préservation du typage

Après avoir montré que la traduction $\mathcal{T}$ préserve la sémantique nous montrons qu'elle préserve également le typage. Notons que si un système de réécriture

préserve la sémantique alors il peut ne pas préserver le typage comme, par exemple, cette règle de réécriture qui préserve la sémantique par réduction :

$MN \rightsquigarrow$ `let` $f(x:T)$ `=` Mx `in` $f(N)$

où il faut pour que le typage soit préservé que le type de N soit un sous-type de T.

Nous allons suivre le même scénario que la preuve précédente, en définissant trois lemmes donnant la préservation du typage pour les trois réécritures $\rightarrow_{\mathcal{S}}$,$\rightarrow_{\mathcal{W}}$ et $\rightarrow_{\mathcal{F}}$ et qui se servent du lemme de substitution suivant :

Lemme 4.2.8 (dit de substitution) *:*

Si $\Gamma \vdash e : t_1$ *et* $\Gamma \vdash E[e]: t_2$ *alors* $\Gamma, x/t_1 \vdash E[x]: t_2$.

> ***Preuve :***
> C'est une propriété de $\mathbb{C}$Duce [Fri04c]. □

Lemme 4.2.9 (Préservation du typage pour $\rightarrow_{\mathcal{S}}$) *:*

Si $\Gamma \vdash q : t$ *et* $q \rightarrow_{\mathcal{S}}^* q_s$ *alors* $\Gamma \vdash q_s : t$

Si à partir de l'environnement de typage Γ*, on obtient que la requête* q *a pour type* t *alors la requête* q_s *aura aussi le type* t*. Il y a alors préservation du typage.*

> ***Preuve :***
> Prouvons par induction sur le nombre de pas d'application de $\rightarrow_{\mathcal{S}}$ que le typage est préservé.
>
> **Cas de base :** La requête q est une requête sans projection, alors $q = \lfloor q \rfloor_{\mathcal{T}}$, donc préservation immédiate du typage.
>
> **Cas inductif :** Soient $\Gamma \vdash q : t_0$ et $q \rightarrow_{\mathcal{S}} q' \rightarrow_{\mathcal{S}}^n q_s$. Montrons que le type de q' est bien t_0 (d'où le résultat suit par hypothèse d'induction).
> Analysons les différents cas S1 et S2.
> Appliqué à S1.
>
> S1 : `select` $E[\bar{e}/t]$ `from` f `where` $c \rightarrow_{\mathcal{S}}$ `select` $E[x]$ `from` f, x `in` $[\bar{e}/t]$ `where` c, $x \notin bv(f)$
>
> D'après la règle de typage du `select from where` (donnée dans la figure 3.2 page 55) :

$$\frac{\Gamma \vdash e_1 : [t_1+] \quad \Gamma,(t_1/p_1),\ldots,(t_{i-1}/p_{i-1}) \vdash e_i : [t_i+] \quad \Gamma,(t_1/p_1),\ldots,(t_n/p_n) \vdash E[\bar{e}/t]: t_0 \quad \Gamma,(t_1/p_1),\ldots,(t_n/p_n) \vdash c : Bool}{\Gamma \vdash \texttt{select } E[\bar{e}/t] \texttt{ from } p_1 \texttt{ in } e_1, \ldots, p_n \texttt{ in } e_n \texttt{ where } c : [t_0*]}$$

Soit t_{n+1}, tel que $\Gamma,(t_1/p_1),\ldots,(t_n/p_n) \vdash \bar{e}/t : t_{n+1}$. L'expression $\bar{e}/t$ peut être typée dans cet environnement de typage, car la règle de typage de q nous donne $\Gamma,(t_1/p_1),\ldots,(t_n/p_n) \vdash E[\bar{e}/t]: t_0$.

D'après le lemme de substitution 4.2.8 en déduit que $\Gamma,(t_1/p_1),\ldots,(t_n/p_n),(t_{n+1}/x) \vdash E[x]: t_0$

On en déduit donc le type de q' :

$$\frac{\Gamma \vdash e_1 : [t_1+] \quad \Gamma,(t_1/p_1),\ldots,(t_{i-1}/p_{i-1}) \vdash e_i : [t_i+] \quad (t_1/p_1),\ldots,(t_n/p_n) \vdash [\bar{e}/t] : [t_{n+1}+] \quad \Gamma,(t_1/p_1),\ldots,(t_n/p_n),(t_{n+1}/x) \vdash E[x]: t_0 \quad \Gamma,(t_1/p_1),\ldots,(t_n/p_n),(t_{n+1}/x) \vdash c : Bool}{\Gamma \vdash \texttt{select E}[x] \texttt{ from } p_1 \texttt{ in } e_1, \ldots, p_n \texttt{ in } e_n, x \texttt{ in } \bar{e}/t \texttt{ where } c : [t_0*]}$$

Ainsi si $\Gamma \vdash q : t$ alors on que $\Gamma \vdash q' : t$, pour l'étape S1. L'étape S2 se prouve de la même manière. □

Lemme 4.2.10 (Préservation du typage pour $\rightarrow_{\mathcal{W}}$) *: Si* $\Gamma \vdash q : t$ *et* $q \rightarrow_{\mathcal{W}}{}^* q_w$ *alors* $\Gamma \vdash q_w : t$

Preuve :

Identique à la preuve du lemme 4.2.9.

□

Lemme 4.2.11 (Préservation du typage pour $\rightarrow_{\mathcal{F}}$) *: Si* $\Gamma \vdash q : t$ *et* $q \rightarrow_{\mathcal{F}}{}^* q_f$ *alors* $\Gamma \vdash q_f : t$

Preuve :

Prouvons par induction sur le nombre de pas d'application de $\rightarrow_{\mathcal{F}}$ que le typage est préservé.

Cas de base : La requête q est une requête sans projection, alors $q = \lfloor q \rfloor_{\mathcal{T}}$, donc préservation immédiate du typage.

Cas inductif : Soient $\Gamma \vdash q : t_0$ et $q \rightarrow_{\mathcal{F}} q' \rightarrow_{\mathcal{F}}{}^n \bar{q}$. Montrons que le type de q' est bien t_0 (d'où le résultat suit par hypothèse d'induction). Analysons les différents cas F1 et F2.

F1 : select $\bar{e}_0$ from $\bar{f}$, p in $E[\bar{e}/t]$, f where $\bar{e} \rightarrow_{\mathcal{F}}$ Si le type de $\bar{e}$ est un sous-type de [Any] alors select $\bar{e}_0$ from $\bar{f}$, <_ ..>[($x :: t$|_)*] in $\bar{e}$, p in $E[x]$, f where $\bar{e}$, $x \notin bv(\bar{f}) \cup bv(f)$ sinon select $\bar{e}_0$ from $\bar{f}$, [(<_ ..>[($x :: t$|_)*] | x ::'nil)*] in [$\bar{e}$] , p in E[flatten(x)] , f where $\bar{e}$, $x \notin bv(\bar{f}) \cup bv(f)$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : [t_1+] \\ \Gamma, (t_1/p_1), \ldots, (t_{j-1}/p_{j-1}) \vdash e_j : [t_j+] \quad \Gamma, (t_1/p_1), \ldots, (t_n/p_n) \vdash e : t_0 \\ \Gamma, (t_1/p_1), \ldots, (t_n/p_n) \vdash c : Bool\end{array}}{\Gamma \vdash \texttt{select } e \texttt{ from } p_1 \texttt{ in } e_1, \ \ldots, p_i \texttt{ in } E[\bar{e}/t], \ldots, p_n \texttt{ in } e_n \texttt{ where } c : [t_0*]}$$

On a donc que $\Gamma, (t_1/p_1), \ldots, (t_{i-1}/p_{i-1}) \vdash E[\bar{e}/t] : [t_i+]$.

Puisque $\Gamma, (t_1/p_1), \ldots, (t_{i-1}/p_{i-1})(t_i/$<_ ..>[($x :: t$|_)*]$) \vdash x : [t*]$, et que $\Gamma, (t_1/p_1), \ldots, (t_{i-1}/p_{i-1}) \vdash \bar{e}/t : [t*]$. d'après le lemme de substitution 4.2.8, on déduit que $\Gamma, (t_1/p_1), \ldots, (t_{i-1}/p_{i-1})(t_i/$<_ ..>[($x :: t$|_)*]$) \vdash E[x] : [t_i+]$.

On a donc au final le type de q' :

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : [t_1+] \\ \Gamma, (t_1/p_1), \ldots, (t_{j-1}/p_{j-1}) \vdash e_j : [t_j+] \quad \Gamma, (t_1/p_1), \ldots, (t_n/p_n) \vdash e : t_0 \\ \Gamma, (t_1/p_1), \ldots, (t_n/p_n) \vdash c : Bool\end{array}}{\Gamma \vdash \texttt{select } e \texttt{ from } p_1 \texttt{ in } e_1, \ \ldots, <_..>[(x :: t|_)*] \texttt{ in } \bar{e}, \ p_i \texttt{ in } E[x], \ \ldots, p_n \texttt{ in } e_n \texttt{ where } c : [t_0*]}$$

On a prouvé le premier cas de la règle F1, le deuxième cas se déduit de la même façon. Également pour le cas F2. On a donc montré par induction que la réduction $\rightarrow_{\mathcal{F}}$ préservait le typage.

□

Théorème 4.2.12 (Préservation du typage de $\mathcal{T}$)
Si $\Gamma \vdash q : t$ *alors* $\Gamma \vdash \lfloor q \rfloor_{\mathcal{T}} : t$

Si à partir de l'environnement de typage Γ*, on obtient que la requête* q *a pour type* t *alors la requête résultant de la traduction* $\mathcal{T}$ *sur la requête* q*, aura le même type* t*.*

Preuve :
Immédiate en appliquant sur q les lemmes 4.2.9, 4.2.10 et 4.2.11. □

4.2.4 Exemple

Le résultat de $\mathcal{T}$ est une requête $\mathbb{C}$QL dans laquelle toutes les projections ont été retirées. Nous verrons dans la section 5.2 que cela a un impact fort sur le temps d'exécution des requêtes avec projections. La requête de la figure 4.1 sera réécrite par $\mathcal{T}$ dans la requête de la figure 4.8

```
<bib>
select <book year=y>[t]
from <_ ..>[(yb::Book|_)*] in [biblio],
     b in yb,
     <_>[(yp::Publisher|_)*] in [b],
     p in yp,
     <_ ..>[(yt::Title|_)*] in [b],
     t in yt,
     <_ year=y>_ in [b]
where (p = <publisher>"Addison-Wesley") and (int_of(y)>>1990)
```

FIG. 4.8 – Requête après la traduction $\mathcal{T}$

Cette traduction peut s'utiliser pour des requêtes contenant les deux types de déconstructeurs, car ne réécrivant que les projections sur élément et attribut, tout en gardant intacts les motifs, ce qui est un point important pour l'objectif que nous nous étions fixés pour la définition du langage de requête $\mathbb{C}$QL.

Modèles de coûts. Il n'y a qu'une seule réécriture possible par $\mathcal{T}$ pour une requête $\mathbb{C}$QL donnée, car les règles S, s'évaluent avant les règles W, qui s'évaluent avant les F. Nous pourrions modifier l'ordre des motifs déduits, et ainsi imaginer qu'en ayant connaissance de taux de sélectivité de certains motifs, on puisse appliquer les modèles de coûts décrits dans la littérature des bases de données, en vue de réduire la taille des données intermédiaires; et réduire ainsi de manière conséquente le temps de calcul de la requête résultante. Cette possibilité d'optimisation n'a pas été étudiée dans cett thèse, et pourra faire l'objet d'un travail futur. Cependant nous proposons une autre méthode inspirée de la descente des sélections que nous transposons dans notre travail dans la section suivante.

4.3 Autres optimisations

Dans cette section nous adaptons des techniques classiques d'optimisations de base de données dans le cas du filtrage par motifs. De telles optimisations évaluent, comme il est très courant dans le monde des bases de données, des conditions dès qu'elles sont demandées. Ceci dans le but de réduire la taille (en mémoire) des données intermédiaires qui contribuent à la construction du résultat.

Nous procédons en trois étapes :

4.3.1 Transformation en motifs d'une partie de la condition

La première étape consiste à mettre la condition de la clause `where` en forme normale conjonctive, et de réécrire chaque clause[5], qui peut l'être, en motifs dans la clause `from`.

Cette première étape est définie par les règles de réécritures suivantes :

`select` e `from` f `where` $c \rightsquigarrow$ `select` e `from` f,$\Theta^1(CNF(c))$ `where` $\Theta^2(CNF(c))$

où CNF(c) est la forme normale conjonctive de la condition c, $\Theta^1(c)$ représente les clauses de c qui peuvent être exprimées par un motif et ainsi remontées dans la clause `from`, et $\Theta^2(c)$ les clauses de c qui restent dans la clause `where`. Formellement, Θ^1 et Θ^2 sont les première et seconde projections de la fonction Θ définie ci-après :

Définition 4.3.1 *Soient i et v des valeurs* $\mathbb{C}$Duce

- $\Theta(v$=$e) = (v$ `in` $[e]$, `'true`$)$
- $\Theta($`member(`v`,`e`)`$) = ($ `[ _`* v `_*]` `in` $[e]$, `'true`$)$
- $\Theta($`count(`e`) =` $i) = ($ `[ _`^{i}`]` `in` $[e]$, `'true`$)$
- $\Theta($`count(`e`) >>` $i) = ($ `[ _`^{i} `_+]` `in` $[e]$, `'true`$)$
- $\Theta($`count(`e`) >=` $i) = ($ `[ _`^{i} `_*]` `in` $[e]$, `'true`$)$
- $\Theta($`count(`e`) <<` $i) = ($ `[(_ ?)`$^{i-1}$`]` `in` $[e]$, `'true`$)$
- $\Theta($`count(`e`) <=` $i) = ($ `[(_ ?)`^{i}`]` `in` $[e]$, `'true`$)$
- $\Theta(e$ `>=` $i) = ($ i`--*` `in` $[e]$, `'true`$)$
- $\Theta(e$ `>>` $i) = ($ $[\![i+1]\!]$`--*` `in` $[e]$, `'true`$)$
- $\Theta(e$ `<=` $i) = ($ `*--`i `in` $[e]$, `'true`$)$
- $\Theta(e$ `<<` $i) = ($ `*--`$[\![i-1]\!]$ `in` $[e]$, `'true`$)$
- $\Theta(c_1$ `and` c_2 `and ...and` $c_n) =$
  $((\Theta^1(c_1), \Theta^1(c_2), \ldots, \Theta^1(c_n))$, $\Theta^2(c_1)$ `and` $\Theta^2(c_2)$ `and ...and` $\Theta^2(c_n))$
- $\Theta(c) = (\epsilon, c)$ *(si aucune des règles précédentes ne s'applique)*

Nous utilisons la notation `_`^{i} pour dénoter la juxtaposition de i occurrences de "`_`". Par exemple, la troisième règle indique que la contrainte `count(`e`) =` 3 peut être filtrée de manière équivalente sur e par le motif `[ _ _ _ ]` (i.e. `[ _`^{3}`]`). Nous utilisons aussi la notation[6] $[\![f(i)]\!]$ pour la constante résultat de $f(i)$.

[5]Une clause d'une condition en forme normale conjonctive est une condition disjonctive. La condition initiale étant la conjonction de toutes ces clauses.

[6]Le motif $[\![i+1]\!]$`--*` est équivalent au motif i`--*` $\backslash i$. Nous utilisons les $[\![\ ]\!]$ pour des

Si nous appliquons ces règles de réécriture sur la requête[7] de la figure 4.8, alors nous pouvons utiliser la première règle de la définition 4.3.1 pour exprimer la clause `p = <publisher>"Addison-Wesley"` par un motif tel qu'illustrée par la figure 4.9.

Ces règles sont loin d'être exhaustives, mais sont très utiles pour les requêtes très courantes avec ce type de conditions.

```
<bib>
select <book year=y>[t]
from <_ ..>[(yb::Book|_)*] in [biblio],
     b in yb,
     <_ ..>[(yp::Publisher|_)*] in [b],
     p in yp,
     <_ ..>[(yt::Title|_)*] in [b],
     t in yt,
     <_ year=y>_ in [b],
     <publisher>"Addison-Wesley" in [p],
     1991--* in [ int_of(y) ]
```

FIG. 4.9 – Application de Θ sur la requête après la traduction $\mathcal{T}$

4.3.2 Consolidation des motifs

La deuxième étape consiste en l'élimination des déclarations intermédiaires inutiles qui sont généralement générées par les déclarations précédentes.

`select` $e_\circ$ `from` f_1, x `in` e, f_2, p_1 `in` $[x]$,f_3,...,p_n `in` $[x]$, f_m `where` c

$\rightsquigarrow$ `select` $e_\circ$ `from` f_1, x&p_1&...&p_n `in` e,f_2 , f_3,..., f_m `where` c

tel que $var(p_1) \cap \cdots \cap var(p_n) = \varnothing$

L'idée est donc de regrouper plusieurs lignes en une seule, si dans la première nous déclarons la variable x comme motif, et si dans des motifs suivants nous l'utilisons pour exprimer un motif sur cette seule valeur ($[x]$). La condition $var(p_1) \cap \cdots \cap var(p_n) = \varnothing$ est importante pour éviter que des motifs aient des variables en commun. Ce genre de cas est problématique :

```
select [z y]
from x in [[1 2 3 4],
     [y::1--2* _*] in [x],
```

raisons de clarté.

[7] La condition étant déjà en forme normale conjonctive.

```
     z in [y],
     [y::1--3* _*] in [x]
```

Cette requête qui n'a pas de sens calcule dans le premier y la séquence `[1 2]` et la pose dans la variable z. Et dans le second y la séquence `[1 2 3]`. Cette requête va être traduite par cette consolidation des motifs en :

```
select [z y]
from x in [[1 2 3 4],
     [y::1--2* _*]&[y::1--3* _*] in [x],
     z  in [y]
```

Ce qui pose un problème, car dans cette exemple, la variable y ne devrait capturer qu'une seule séquence. Néanmoins ce genre de cas n'arrive pas car la traduction $\mathcal{T}$ assure que chaque nouvelle variable introduite est fraîche.

Dans l'exemple précédent, la consolidation des motifs s'applique sur les variables p et b. On aura donc la requête de la figure 4.10.

```
<bib>
select <book year=y>[t]
from <_ ..>[(yb::Book|_)*] in [biblio],
     b&<_ ..>[(yp::Publisher|_)*]
      &<_ ..>[(yt::Title|_)*]&<_ year=y>_ in yb,
     p&<publisher>"Addison-Wesley" in yp,
     t in yt,
     1991--* in [ int_of(y) ]
```

FIG. 4.10 – Consolidation des motifs

Nous pouvons noter que des variables sont alors définies sans qu'elles soient utilisées dans la suite, comme b et p. $\mathbb{C}$Duce est capable de le détecter ; les retirer est simple et nous ne l'étudierons pas.

4.3.3 Remontée des sélections

La troisième et dernière étape est une technique classique de remontée des sélections aussi proche que possible de la déclaration des variables sur lesquelles portent les clauses de la condition. La manière de le faire est de modifier la sémantique opérationnelle du `select from where` en y ajoutant l'opérateur `if then else`. L'exemple précédent ne s'y prête pas, mais

nous pourrions imaginer par exemple une condition initiale comme celle-ci : `where (p = <publisher>"Addison-Wesley") and ((int_of(y)>>1990) or (int_of(y)=1980))`. Cette condition est en forme normale conjonctive, et la deuxième clause ne peut-être traitée par la réécriture de "mise en motifs de certaines conditions", car il n'existe pas de règle portant sur des conditions disjonctives.

Cette requête après la traduction $\mathcal{T}$, la remontée des sélections, puis la consolidation des motifs aurait donné la requête de la figure 4.11.

```
<bib>
select <book year=y>[t]
from <_ ..>[(yb::Book|_)*] in [biblio],
     b&<_ ..>[(yp::Publisher|_)*]
      &<_ ..>[(yt::Title|_)*]&<_ year=y>_ in yb,
     p&<publisher>"Addison-Wesley" in yp,
     t in yt,
where ((int_of(y)>>1990) or (int_of(y)=1980))
```

FIG. 4.11 – Requête avec condition disjonctive

Dans ce cas, puisque la condition a pour ensemble de variables $\{y\}$, cette condition sera évaluée en dernier juste avant la création de l'élément $<$`book`$>$, alors que la variable y est capturée bien avant. Il suffit de placer l'opérateur `if then else` avec la condition juste après la déclaration de toutes les variables que comporte la clause `where`. Ce qui réduit la taille des données intermédiaires suivant le taux de sélectivité de la condition. Nous étudierons cet impact sur les performances dans le chapitre suivant. Ainsi, on modifie la sémantique opérationnelle d'une requête $\mathbb{C}$QL, en déplaçant l'opérateur `if then else`. Sur l'exemple de la figure 4.11, cette remontée de sélection nous donnera le code $\mathbb{C}$Duce de la figure 4.12

Ce genre d'optimisation n'a rien de nouveau, et correspond aux optimisations classiques logiques du modèle relationnel.

4.4 Conclusion

Nous avons apporté à $\mathbb{C}$QL la traduction $\mathcal{T}$ qui réécrit toutes les projections d'une requête non-imbriquée en motifs, et nous avons prouvé que la sémantique est préservée. Nous avons aussi apporté une transformation de conditions en motifs, et une règle de consolidation de motifs qui s'applique à la traduction $\mathcal{T}$. Nous avons

```
<bib>
transform [biblio] with
<_ ..>[(yb::Book|_)*] ->
   transform yb with
   b&<_ ..>[(yp::Publisher|_)*]&<_ ..>[(yt::Title|_)*]&<_ year=y>_ ->
      if ((int_of(y)>>1990) or (int_of(y)=1980))
      then transform yp with
           p&<publisher>"Addison-Wesley" ->
              transform yt with
              t -> [<book> year=y>[t] ]
      else []
```

FIG. 4.12 – Requête avec condition remontée

aussi transposé une technique logique courante en base de données qui consiste à remonter les projections. Nous allons étudier dans le chapitre suivant les impacts de ces optimisations sur des jeux de tests, ce qui nous permettra de les valider.

Chapitre 5

Performances

Sommaire

5.1 Motivations 89

5.2 Approche *Jeux de tests* 90

5.2.1 Sur les *XML Query Use Cases* 91

5.2.2 Sur XMark 103

5.3 Approche *micro-benchmarks* 107

5.3.1 Présentation des requêtes 110

5.3.2 Résultats de $\mathbb{C}$QL 115

5.3.3 Quelques performances comparées 119

5.4 Conclusion 121

5.1 Motivations

Nous avons cherché tout d'abord à comparer les différents déconstructeurs sur différents documents XML en suivant l'approche *jeux de tests* qui consiste à comparer un ensemble de requêtes concrètes sur un document correspondant à une utilisation courante. Nous avons considéré en premier les « XML Query Use Cases » qui se présentent comme une série de requêtes qu'un utilisateur doit pouvoir écrire avec un langage de requêtes pour interroger des documents XML ; ainsi que sur XMark[SWK+01, SWK+02] qui donne un ensemble de benchmarks utilisant pleinement les caractéristiques et les possibilités de XML. Nous avons aussi étudié ce que la traduction et les algorithmes présentés dans le chapitre précédent apportaient en gain de temps dans ces séries de jeux de tests.

La deuxième approche dite "*micro-benchmarks*" sera suivie dans la section suivante. Elle consiste à comparer des requêtes sur des documents non concrets, en isolant les composantes communes de requêtes dans le but d'en déduire leurs

performances ; car des performances fortes ou faibles d'un composant peuvent être cachées dans des requêtes complexes considérées dans leur ensemble, notamment dans le cas de jeux de tests.

Dans ces études de performance de $\mathbb{C}$QL, on rappelle que $\mathbb{C}$QL est implanté dans $\mathbb{C}$Duce, et qu'il ne possède pas de stockage de données via un SGBD, les requêtes écrites en $\mathbb{C}$QL sont traduites en du code $\mathbb{C}$Duce. Ce code sera exécuté à partir de fichier stockés sur disque.

5.2 Approche *Jeux de tests*

Les mesures de performance ont été exécutées [BCM04, BCM05] sur un PENTIUM IV à 3,2 GHz avec 1 Go de RAM sur une FREEBSD 5.2.1, avec $\mathbb{C}$Duce 0.2. Nous avons comparé pour chaque requête, différentes "écritures" de $\mathbb{C}$QL utilisant différents déconstructeurs.

- La première requête n'utilise uniquement que la navigation par chemins, appelée dans la suite $\mathbb{C}\text{QL}_X$.
- La deuxième est la requête résultant de la première par l'application de l'algorithme de traduction $\mathcal{T}$ présenté dans le chapitre 4, et de la mise en motifs de la condition et de la consolidation des motifs, appelé $\mathbb{C}\text{QL}_P$.
- La troisième est la requête $\mathbb{C}\text{QL}_X$ avec application de l'algorithme de remontée de sélection présenté dans la section 4.3.3, appelé ${\mathbb{C}\text{QL}_X}^{opt}$.
- La quatrième est la requête $\mathbb{C}\text{QL}_P$ avec remontée de la sélection, appelé ${\mathbb{C}\text{QL}_P}^{opt}$.
- La dernière est la requête écrite directement avec des motifs ad-hoc, qui correspond à une requête écrite par un utilisateur averti ayant connaissance de la DTD.

Nous comparerons aussi les temps d'exécution des requêtes $\mathbb{C}$QL avec des requêtes similaires écrites en XQUERY sur différentes implantations.

Nous avons choisi pour nos tests, de manière légèrement arbitraire, de n'étudier que les cinq premières requêtes Q1, Q2, Q3, Q4 et Q5 de la partie XMP des « *XML Query Use Cases* »[1][CFF+03]. Le document XML qui sert de base à ces requêtes représente une bibliographie qui est présentée à la figure 2.1, ainsi que la DTD associée à la figure 2.3. Nous étudierons aussi quelques requêtes choisies de XMARK.

Ces expériences ont été réalisées au début de l'année 2005, nous avons comparé $\mathbb{C}$QL à différentes implantations de XQUERY du moment libres d'utilisation : GA-

[1] Qui en comportent plus de 70.

LAX [Bel], QIZX [Fra], et QEXO [Bot, Bot04]. GALAX est l'implantation de XQUERY la plus complète au niveau des spécifications de "*XML Query*"[BCF+01]. Elle est écrite en OCaml[oca]. QIZX/OPEN (version 0.3) est une implantation en JAVA, libre d'utilisation, des spécifications de XQUERY développée pour une distribution commerciale et qui a été spécialement conçue pour une exécution efficace sur de grandes bases de données. QEXO est une implantation partielle de XQUERY qui réalise des bonnes performances en compilant directement les requêtes en du bytecode JAVA par l'utilisation du framework KAWA (version 1.6.98).

5.2.1 Sur les *XML Query Use Cases*

Ces cas d'utilisation ont été apportés par le groupe de travail "*XML Query*" du W3C pour montrer les différentes et importantes applications que l'on attend d'un langage de requêtes pour XML[FSW+99].

La grande majorité des requêtes sont exprimables en $\mathbb{C}$QL grâce à l'utilisation des motifs : exceptées celles qui font appel à l'opérateur "//" pour des mêmes éléments imbriqués à différents niveaux. Pour ce qui est de $\mathbb{C}$QL avec projections, comme nous avons pris le choix délibéré de ne pas contenir tout XPATH, les requêtes ne pouvant être exprimées sans l'utilisation de fonctions ad-hoc sont plus nombreuses.

Comme l'algorithme de traduction $\mathcal{T}$ ne considère pas l'opération de projection sur descendants ("//"), nous avons choisi de traduire ces dernières avec des projections simples sur éléments, dans les requêtes XQUERY et $\mathbb{C}\mathrm{QL}_X$: ce qui est permis par la DTD de la figure 2.3 car n'utilisant pas d'entités récursives (comme dans la DTD de la figure 3.11). Ce qui donne un avantage aux requêtes $\mathbb{C}\mathrm{QL}_X$ par rapport aux autres requêtes, car ces projections plus fines seront mieux évaluées que les projections sur descendants par $\mathbb{C}$Duce.

Les cinq premières requêtes représentent les principales utilisations d'un langage de requêtes, nous nous y sommes donc restreints. La requête Q1 exécute une simple sélection. Les requêtes Q2 et Q3 sont des requêtes de reconstruction, chacune des deux donne la liste de la bibliographie entière, de telle sorte que la première renvoie une liste désimbriquée de paires titre/auteur (chaque paire étant contenue dans un élément `<result>`), la seconde renvoie le titre et l'ensemble des auteurs, groupés dans un élément `<result>`. Pour chaque auteur de la bibliographie, la requête Q4 donne la bibliographie personnalisée de chaque auteur, groupée dans un élément `<result>`. Et la requête Q5 exécute une jointure entre deux documents, de manière à faire la liste pour chaque livre des différents prix que l'on

peut trouver dans chaque source.

	Taille	Taille2	flt$_{CQL}$	$\mathbb{C}$QL$_X$	$\mathbb{C}$QL$_X^{opt}$	$\mathbb{C}$QL$_P$	$\mathbb{C}$QL$_P^{opt}$	$\mathbb{C}$QL	Qizx	Qexo
Q1	36 Ko		0.01	0.01	0.01	0.02	0.01	0.01	0.45	0.60
Q1	1.8 Mo		0.23	0.26	0.25	0.26	0.26	0.24	0.76	1.01
Q1	14 Mo		1.90	2.00	1.99	1.98	2.07	1.93	2.18	2.89
Q1	35 Mo		4.79	5.13	5.04	5.03	5.24	4.90	4.44	5.80
Q2	36 Ko		0.01	0.01	0.01	0.01	0.01	0.01	0.46	0.61
Q2	1.8 Mo		0.24	0.26	0.26	0.25	0.25	0.25	1.00	1.04
Q2	14 Mo		1.87	2.06	2.06	2.01	2.01	1.99	3.77	3.55
Q2	35 Mo		4.74	5.27	5.27	5.14	5.14	5.08	8.16	7.79
Q3	36 Ko		0.01	0.01	0.01	0.01	0.01	0.01	0.47	0.60
Q3	1.8 Mo		0.24	0.25	0.26	0.25	0.25	0.25	0.99	1.03
Q3	14 Mo		1.90	2.03	2.02	2.01	2.02	2.01	3.66	3.27
Q3	35 Mo		4.81	5.18	5.18	5.14	5.14	5.13	7.90	6.86
Q4	36 Ko		0.01	0.05	0.05	0.05	0.05	0.05	0.53	
Q4	70 Ko		0.02	0.17	0.17	0.14	0.14	0.14	0.68	
Q4	144 Ko		0.02	0.61	0.61	0.52	0.52	0.49	1.17	
Q4	575 Ko		0.09	10.73	10.73	9.94	9.94	8.63	10.97	
Q4	1.8 Mo		0.24	113.01	113.01	89.31	89.31	88.70	104.12	
Q5	36 Ko	535 Ko	0.08	1.69	0.79	1.17	0.71	0.54	4.44	27.88
Q5	144 Ko	43 Ko	0.03	0.52	0.24	0.38	0.24	0.17	1.79	9.31
Q5	575 Ko	171 Ko	0.11	7.87	3.49	5.92	3.34	2.46	20.74	127.39
Q5	1.8 Mo	535 Ko	0.31	78.27	36.54	53.25	31.04	22.93	197	>1h
Q5	3.5 Mo	535 Ko	0.55	157.70	72.28	105.38	62.24	45.02	392	

(flt = temps de chargement du fichier, la colonne donne la taille du deuxième document pour la jointure.)

FIG. 5.1 – Résumé des résultats des tests sur les requêtes des « *XML Query Use Cases* »

Les résultats de nos tests sont présentés dans le tableau de la figure 5.1, dans lesquels nous n'avons pas répertorié les temps d'exécution pour GALAX ; nous n'avons pas exécuté tous les tests avec GALAX car les premières expériences montraient clairement que les temps d'exécution de GALAX avaient plusieurs ordres de grandeurs en plus que QIZX et QEXO. Les mesures ont été exécutées pour chaque requête sur des documents générés aléatoirement de tailles différentes (exprimées en kilo-octets). Nous avons aussi essayé de regarder l'impact du taux de sélectivité de différentes requêtes sur les temps d'évaluation. Chaque test a été répété plusieurs fois, et les moyennes des temps d'exécution (en secondes) de ces résultats ont été reportées sur le tableau. Nous avons distingué le temps de chargement des documents XML (et de typage) pour $\mathbb{C}$Duce dans la colonne nommée "flt" (pour "*file loading time*"), du temps global d'exécution. Les temps d'exécution incluent le temps d'inférence de types et le temps de chargement (pour les différentes versions $\mathbb{C}$QL). Dans ce tableau sont donnés, en outre, les résultats pour $\mathbb{C}$QL, qui correspond à une requête écrite avec motifs par un utilisateur expérimenté. Dans la

suite, nous ne les verrons pas ; ils indiquent que les meilleurs résultats sont obtenus par une utilisation complète des motifs plutôt que par l'utilisation de projections, ce qui valide la transformation des projections en motifs par la traduction $\mathcal{T}$.

En comparant le temps de chargement au temps d'exécution global (ce dernier inclut le premier temps) il est clair que les seules requêtes avec un temps de calcul significatif sont les requêtes Q4 et Q5 (QEXO n'avait pas encore à ce moment implanté la fonction `distinct_values`). Dans ces deux cas les meilleures performances sont obtenues par $\mathbb{C}$QL$_P$. Le cas de QIZX est particulier. Quand QIZX détecte une jointure avec une condition, il génère directement une table de hashage dans le but de s'en servir ensuite comme index[2]. Dans le but de comparer nos moteurs de requêtes, et parce que nous croyons que la génération d'index doit être laissée plutôt à un optimiseur de requêtes annexe (qui n'existe pas encore pour $\mathbb{C}$QL) qu'au compilateur de requêtes, nous avons désactivé la génération d'index de QIZX (en ajoutant à la condition de jointure la condition disjonctive " or false").

Nous allons décrire en détail chaque requête et les tests correspondants. Nous présentons les codes en XQUERY, $\mathbb{C}$QL$_X$,$\mathbb{C}$QL$_P$, et $\mathbb{C}$QL. Les codes $\mathbb{C}$QL utilisés pour ces tests n'incluent pas les instructions $\mathbb{C}$Duce pour charger ces fichiers à partir des documents XML, qui sont représentés par les variables `biblio`, et `bstore2` pour la requête Q5. Et nous commenterons les résultats à l'aide des graphiques générés avec pour abscisses la taille des documents, et en ordonnées le temps d'exécution de la requête pour $\mathbb{C}$QL$_X$, $\mathbb{C}$QL$_P$, et optimisation de la remontée de sélection ($\mathbb{C}$QL$_X^{opt}$ et $\mathbb{C}$QL$_P^{opt}$), QIZX et QEXO.

5.2.1.1 La requête Q1

Les « XML Query Use Cases » demandent à ce que la requête Q1 renvoie la liste des livres publiés par Addisson-Wesley après l'année 1991, en incluant leur année de publication et leur titre. Les codes des requêtes sont présentés dans la figure 5.3. Les courbes dans la figure 5.2.

Sur cette série de tests, $\mathbb{C}$QL$_P$ est plus rapide que QEXO, et à un temps comparable à QIZX, quoique ce dernier soit meilleur. Les différentes versions de $\mathbb{C}$QL$_P$ ont un temps d'exécution similaire, et les légères différences représentent simplement du bruit statistique. Ces différences deviendront évidentes seulement avec les requêtes qui demandent plus de calculs : Q4 et Q5.

[2]Pour des benchmarks peu sélectifs, cela accélère considérablement le temps de calcul et apporte bien sûr de meilleurs performances.

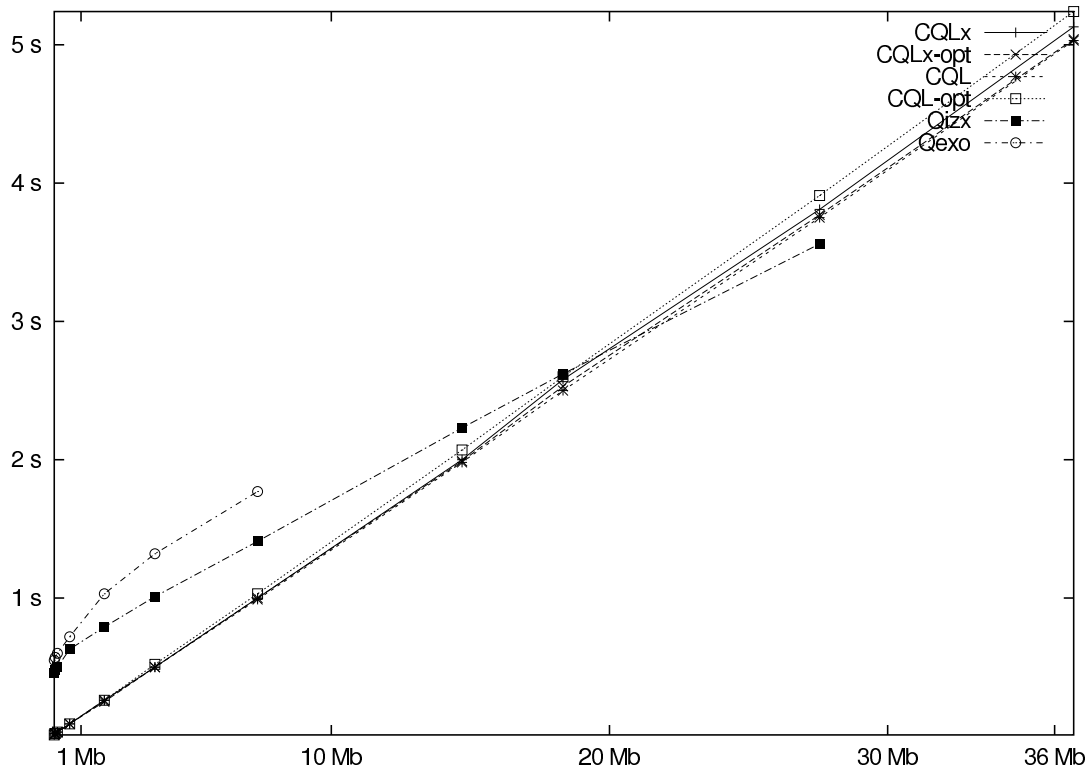


FIG. 5.2 – Temps d'exécution (sans temps de chargement) pour la requête Q1

5.2.1.2 Les requêtes Q_2 et Q_3

La requête Q2 crée une liste plate de tous les paires titre/auteur, telle que chaque paire est comprise dans un élément `<result>`. Les codes des requêtes sont donnés dans la figure 5.4.

La requête Q3 est légèrement différente que la requête Q2. Il s'agit ici de considérer non plus chaque auteur indépendemment des autres, mais l'ensemble des auteurs pour chaque titre, et demande donc à les renvoyer, pour chaque livre de la bibliographie, encapsulés dans un nouvel élément `<result>` (figure 5.5).

Les requêtes Q2 et Q3 sont très simples, et n'ont pas de condition de sélection. Les figures 5.6 et 5.7, montrent des performances quasiment égales bien que $\mathbb{C}\mathrm{QL}_P$ ait de meilleurs temps.

5.2.1.3 La requête Q_4

Pour chaque auteur, la requête Q4 donne la liste des titres des livres auxquels ils ont participés, groupés dans un élément `<result>`. La requête Q4 utilise deux opérateurs génériques : `member` et `distinct-values` qui respectivement testent l'appartenance d'un élément dans une séquence, et qui retirent les occurrences de chaque élément dans une séquence. Ces opérateurs sont nécessaires pour capturer la liste des auteurs sans doublons, et ensuite vérifier qu'ils soient ou non auteur

Requête en XQUERY :

```
<bib>
 {
  for $b in document("http://www.bn.com/bib.xml")/bib/book
  where $b/publisher = ''Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

Requête en $\mathbb{C}$QL$_X$:

```
<bib> select <book year=y>([b]/Title)
from b in [biblio]/<book>_ ,
     p in [b]/<publisher>_ ,
     y in [b]/@year
where (p = <publisher>"Addison-Wesley") and (int_of(y)>>1991);;
```

Requête en $\mathbb{C}$QL$_P$:

```
<bib> select <book year=y>[yt]
from <_ ..>[(yb::Book|_)*] in [biblio],
     <_ year=y>_&<_ ..>[(yt::Title|_)*]&<_ ..>[(yp::Publisher|_)*] in yb,
     <publisher>"Addison-Wesley" in yp,
where  int_of(y)>>1991
```

Requête en $\mathbb{C}$QL :

```
<bib> select <book year=y>[t]
from <bib>[b::Book*] in [doc],
     <book year=y>[t&Title _+ <publisher>"Addison-Wesley";_] in b
where  int_of(y)>>1991
```

FIG. 5.3 – Requête Q1 - Donner les titres et années de publication des livres publiés par "Addison-Wesley" après 1991.

du parcouru. Au moment de ces tests, ces fonctions n'étaient pas implantées dans QEXO, nous n'avons donc pas exécuté ces tests avec. La figure 5.9 montre que l'optimisation de remontée de sélection n'a pas d'effet sur $\mathbb{C}$QL (ce qui est attendu vu que dans cette requête la condition ne peut pas être remontée). La seule optimisation notable est la traduction $\mathcal{T}$ en motifs qui permet d'avoir un gain de temps de 15% par rapport à QIZX.

XQUERY :

```
<results>
 {
  for $b in $biblio/bib/book,
        $t in $b/title,
        $a in $b/author
  return
        <result>
            { $t }
            { $a }
        </result>
  }
</results>
```

$\mathbb{C}$QL$_X$:

```
<results>
select <result>[t a]
from b in [biblio]/Book,
         t in [b]/Title,
        a in [b]/Author ;;
```

$\mathbb{C}$QL$_P$:

```
<results>
select <result>[x2 a]
from <_>[(x1::Book|_)*] in [biblio],
     <_>[(x3::Author|x2::Title|_)*] in x2,
     t in x2,
     a in x3
```

$\mathbb{C}$QL :

```
<results>
select <result>[t a]
from <bib>[b::Book+] in [biblio],
        <book>[t&Title la::Author+ ;_] in b,
         a in la
```

FIG. 5.4 – Requête Q2 - Donner chaque paire titre/auteur pour chaque livre de la bibliographie.

XQUERY :

```
<results>
{
    for $b in $biblio/bib/book
    return
        <result>
            { $b/title }
            { $b/author  }
        </result>
}
</results>
```

$\mathbb{C}$QL$_X$:

```
<results>
select <result> ([b]/Title @ [b]/Author )
from b in [biblio]/Book
```

$\mathbb{C}$QL$_P$:

```
<results>
select<result>(x1 @ x2)
from <_>[(x3::Book|_)*] in [biblio],
        <_>[(x1::Title|_)*] & <_>[(x2::Author|_)*] in x3
```

$\mathbb{C}$QL :

```
<results>
select <result>(t @ a)
from <bib>[b::Book*] in [doc],
     <book ..>[t::Title (a::Author+|_)* ;_] in b
```

FIG. 5.5 – Requête Q3 - Créer pour chaque livre de la bibliographie un nouvel élément `<result>` contenant le titre et les auteurs.

5.2.1.4 La requête Q_5

La requête Q5 est une jointure sur deux documents différents comme montré dans le code de la figure 5.10.

Pour chaque livre présent, et dans le document lié à `biblio` et à `amazon` la requête Q5 donne la liste des livres et de leur prix respectifs. C'est avec la requête Q4 celle qui demande le plus de calculs : le temps de chargement est négligeable par rapport au temps d'exécution. C'est dû au fait que les temps de chargement de chaque fichier sont toujours ajoutés, alors qu'ici les temps d'exécution sont multipliés car cela correspond au "produit cartésien" entre ces documents.

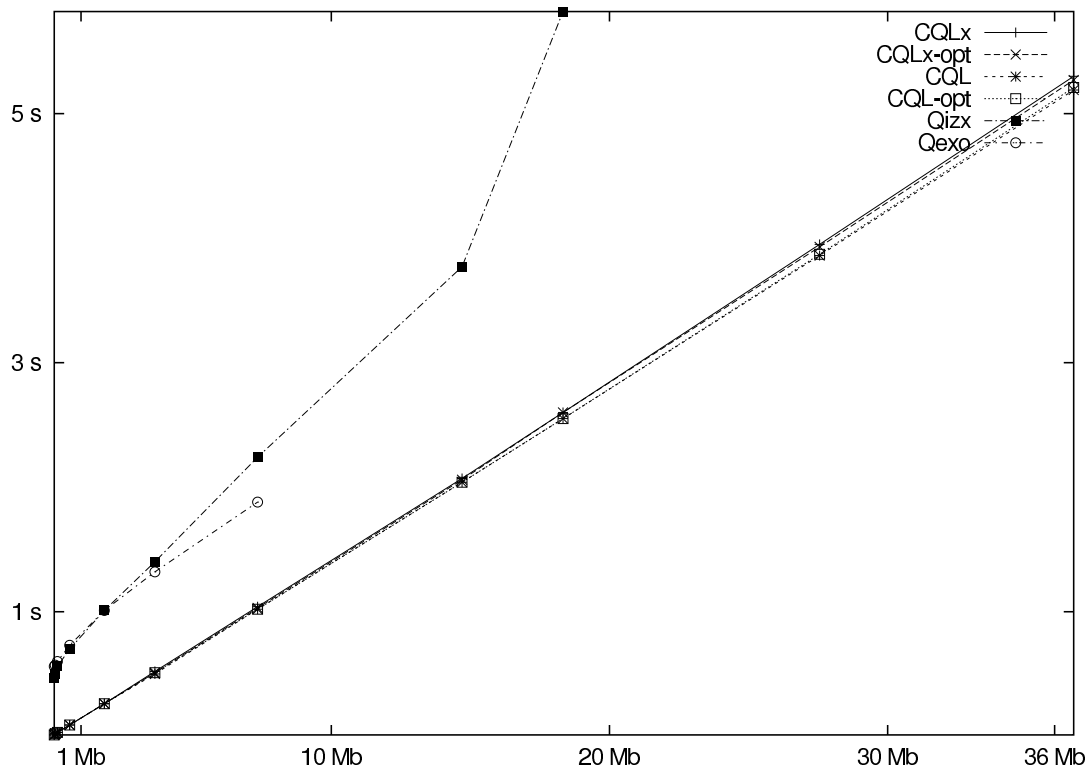


FIG. 5.6 – Temps d'exécution pour la requête Q2

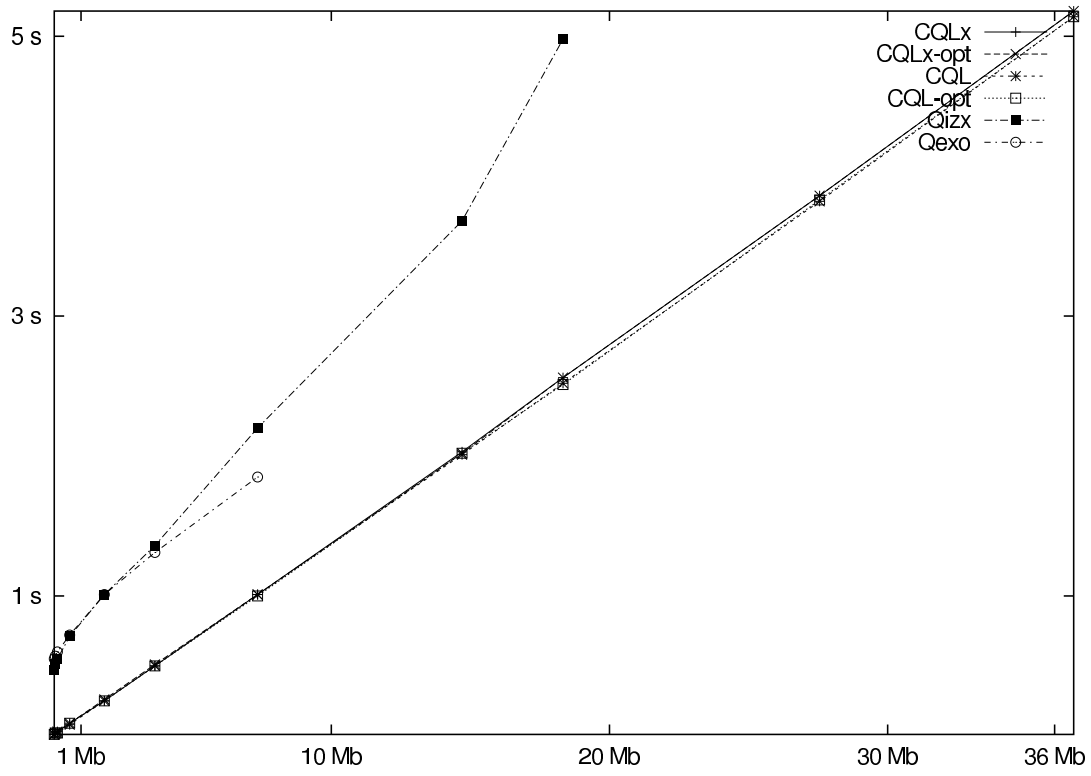


FIG. 5.7 – Temps d'exécution pour la requête for Q3

Le graphique de la figure 5.11 (qui n'inclut pas les temps de QEXO, mais seulement ceux de $\mathbb{C}$QL et de QIZX) est intéressant pour plusieurs points.

Le premier est que $\mathbb{C}\text{QL}_P$ est surpassé par QIZX.

XQUERY :

```
<results>
   {
     for $a in distinct-values($biblio//author)
     return
         <result>
             { $a }
             {
                 for $b in $biblio/bib/book
                 where some $ba in $b/author
                       satisfies
                       deep-equal($ba,$a)
                 return $b/title
             }
         </result>
   }
</results>
```

$\mathbb{C}$QL$_X$:

```
<results>
select <result>([a] @ flatten( select [b]/Title
                               from b in [biblio]/Book
                               where member(a,[b]/Author)))
from a in distinct_values( [biblio]/Book/Author)
```

$\mathbb{C}$QL$_P$:

```
<results>
select <result> ([a] @ flatten( select x3
                                from <_ ..>[(x5::Book|_)*] in [biblio],
                         <_ ..>[(x3::Title|_)]&<_ ..>[(x4::Author|_)*] in x5
                                where member(a,x4) ))
from <_ ..>[(x1::Book|_)*] in [biblio],
     <_ ..>[(x2::Author|_)*] in x1,
     a in distinct_values(x2)
```

$\mathbb{C}$QL :

```
<results>
select <result>([a] @(select t
                      from <bib>[b::Book+] in [biblio],
                           <book>[t&Title a2::Author+ ;_] in b
                      where member(a,a2) ))
from a in distinct_values( select a
                           from <bib>[b::Book+] in [biblio],
                                <book>[Title la::Author+ ;_ ] in b,
                                a in la)
```

FIG. 5.8 – Requête Q4 - Donner pour chaque auteur la liste des titres des livres auxquels ils ont participés, groupés dans une balises `<result>`

.

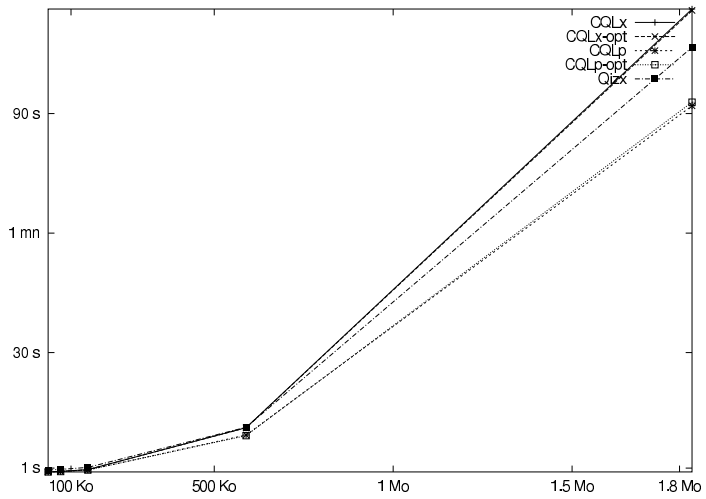


FIG. 5.9 – Temps d'exécution (sans temps de chargement) pour la requête Q4

Le second est que la traduction en motifs de la requête $\mathbb{C}\mathrm{QL}_X$ améliore remarquablement les performances avec un gain de 50% sur les temps d'exécution.

Et le dernier point est que l'optimisation de remontée de la sélection présentée dans la section 4.3.3 apporte un gain de temps notable.

Comme $\mathbb{C}\mathrm{QL}_P$ est partiellement optimisé (car certains motifs "placent" les conditions au bon endroit), les différences après l'optimisation de remontée de sélection sont moins grandes que pour $\mathbb{C}\mathrm{QL}_X$. Néanmoins, elle constitue un gain de 25% sur les performances. C'est pourquoi, bien que ces résultats soient basés sur un jeu de tests très restreint, il nous semble que les optimisations classiques de bases de données peuvent être vraiment utiles même pour du code déjà optimisé par l'utilisation du filtrage par motifs.

Mais il reste que $\mathbb{C}\mathrm{QL}_P$ est largement surpassé par QIZX. La raison principale est que QIZX utilise des optimisations pour quelques jointures précises dont fait partie Q5. Lorsque une jointure possède une seule condition avec égalité, il génère dynamiquement une table de hashage et l'utilise comme index de manière à retrouver directement sur le disque la valeur XML correspondante, au lieu de reparcourir l'ensemble de la bibliographie dans le cas de $\mathbb{C}$QL. Il s'agit de l'algorithme de jointure par hashage dont on trouvera une explication détaillée dans la section 14.4 de [RG02].

Cette technique est redoutablement efficace (et est d'ailleurs utilisée dans la plupart des SGBD traditionnels), mais nous n'avons pas décidé de l'implanter dans $\mathbb{C}$QL. Le gain apporté par l'algorithme de jointure par hashage est quantifiable : il suffit de tricher un petit peu en rajoutant "`or false`" dans la clause `where` (ce qui est équivalent); ainsi cette optimisation n'est plus appliquée car QIZX ne retire pas les tautologies de la condition.

XQUERY :

```
<books-with-prices>
  { for $b in $biblio/book,
        $a in $bstore2/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>
```

$\mathbb{C}$QL$_X$:

```
<books-with-prices>
select <book-with-price>[t1 <price-bstore2> p2 <price-bstore1> p1  ]
from b in [biblio]/Book ,
     y in [b]/@year,
     t1 in [b]/Title,
     e in [amazon]/<entry>_,
     t2 in [e]/Title,
     p2 in [e]/Price,
     p1 in [b]/Price
where t1=t2
```

$\mathbb{C}$QL$_P$:

```
<books-with-prices>
select <book-with-price>[t1 <price-bstore2>(x10) <price-bstore1>(x11) ]
from <_>[(x3::Book|_)*] in [biblio],
     <_ year=y>[(x9::Price|x5::Title|_)*] in x3,
     t1 in x5,
     <_>[(x6::<entry>|_)*] in [bstore2],
     <_>[(x7::Title|x8::Price|_)*] in x6,
     t2 in x7,
     <_>[(x10::_)*] in x8,
     <_>[(x11::_)*] in x9
 where t1=t2
```

$\mathbb{C}$QL :

```
<books-with-prices>
select <book-with-price>[t1 <price-bstore2>p2 <price-bstore1>p1 ]
from <bib>[b::Book*] in [biblio],
        <book year=y&(1991--*)>[t1&Title _* <price>p1] in b,
        <reviews>[e::Entry*] in [bstore2],
        <entry>[t2&Title <price>p2 ;_] in e
where t1=t2
```

FIG. 5.10 – Requête Q5 - Faire un comparatif des prix pour un même titre des livres de la bibliographie et de bstore2.

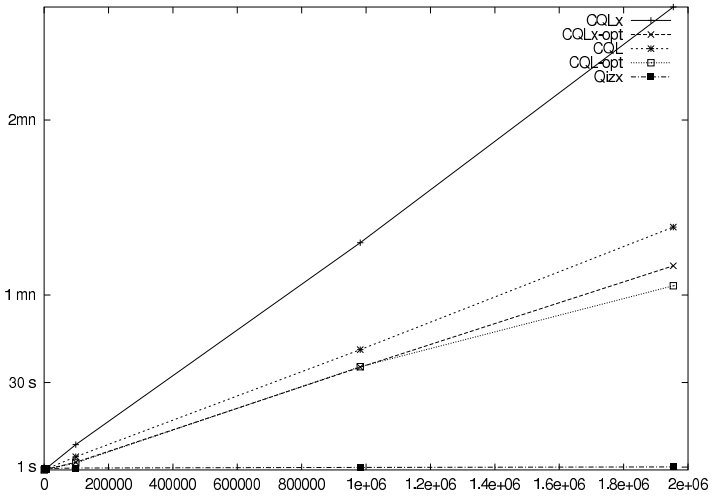


FIG. 5.11 – Temps d'exécution pour la requête Q5 [sans QEXO]
(En abscisses le *produit* des tailles de chaque fichier.)

Taille 1	Taille 2	$\mathbb{C}$QL$_P$	QIZX
36Ko	11Ko	0.05	0.66
72Ko	535Ko	0.12	1.04
144Ko	43Ko	0.38	1.75
575Ko	171Ko	5.92	20.70
1.8Mo	535Ko	53.30	197
3.5Mo	535Ko	105	—

En trichant ainsi, on obtient les temps d'exécution dans le tableau ci-contre, et nous voyons alors que $\mathbb{C}$QL$_P$ devient meilleur que QIZX. En outre le gain obtenu par l'utilisation des tables de hash est très important dans ces tests car nous avons utilisé des documents générés aléatoirement avec un très faible taux de sélectivité[3] (comme cela doit d'ailleurs l'être dans les items d'une bibliographie), ce qui est très profitable pour une jointure par hashage. Nous avons donc joué sur les taux de sélectivité pour obtenir un taux très élevé de 50%[4], alors $\mathbb{C}$QL$_P$ devient meilleur que QIZX, comme on peut le remarquer dans le tableau A de la figure 5.12.

$\mathbb{C}$QL$_P$ est alors environ quatre fois plus rapide que QIZX (les cases vides représentent les cas où la mémoire n'était pas assez importante pour stocker l'ensemble des résultats de la requête).

Dans le cas d'une double auto-jointure nous obtenons les mêmes types de résultats. Le tableau B de la figure 5.12 donne les temps pour celle-ci avec les mêmes fichiers du début avec un taux de sélectivité très faible. La version optimisée de $\mathbb{C}$QL$_P$ est au moins deux fois plus rapide que QIZX. On remarque aussi dans ce cas particulier l'impact que peut avoir les remontées de sélections.

[3]Le titre d'un livre est implicitement (au regard des requêtes proposées) une clef primaire dans le jeu de tests de XQUERY.

[4]Ce qui représente seulement deux titres différents pour l'ensemble de la bibliographie.

5.2.2 Sur XMark

Suivant la suggestion de Ioana Manolescu (qui est une des auteurs du projet XMark), nous avons choisi quatre requêtes qui devraient donner une bonne vue d'ensemble des comportements de $\mathbb{C}$QL. Notre choix s'est porté sur les requêtes détaillées dans [SWK+01] :

- Q1, car il s'agit de la plus simple,
- Q8, qui effectue des traversées horizontales du document avec une complexité croissante,
- Q12, qui teste la capacité à manipuler des résultats intermédiaires de taille conséquente,
- et Q16, effectue des traversées verticales

Les résultats sont résumés dans le tableau de la figure 5.13. Nous n'avons pas exécuté les versions optimisées par l'algorithme de remontée de sélection sur $\mathbb{C}\text{QL}_X$ et $\mathbb{C}\text{QL}_P$, car ces requêtes ne possèdent pas de sélections à remonter. Il n'y a pas les temps de Qexo pour la requête Q12, car l'exécution levait une exception.

Une fois encore si l'on compare les temps de chargement avec les temps d'exécution nous voyons que les requêtes intéressantes sont seulement les requêtes Q8 et Q12. Dans les deux autres ces deux temps sont vraiment proches, ce qui revient à dire qu'elles sont en très grande partie déterminées par le temps de de l'analyseur lexical du document. Il est intéressant de noter que $\mathbb{C}$Duce uti-

Taille 1	Taille 2	$\mathbb{C}\text{QL}_X$	$\mathbb{C}\text{QL}_X^{opt}$	$\mathbb{C}\text{QL}_P$	$\mathbb{C}\text{QL}_P{}^{opt}$	Qizx
36Ko	11Ko	0.09	0.08	0.08	0.06	0.77
72Ko	22Ko	0.31	0.29	0.27	0.25	1.37
144Ko	44Ko	1.28	1.13	1.12	0.98	3.41
293Ko	88Ko	5.87	5.32	5.18	4.59	—
575Ko	175Ko	—	—	—	—	—

Tableau A : La requête Q5 avec un taux de sélectivité de 50%

Taille	$\mathbb{C}\text{QL}_X$	$\mathbb{C}\text{QL}_X^{opt}$	$\mathbb{C}\text{QL}_P$	$\mathbb{C}\text{QL}_P{}^{opt}$	Qizx
36Ko	2.62	0.06	2.07	0.06	0.65
72Ko	21	0.19	17	0.19	1.00
144Ko	199	0.72	148	0.72	2.20
575Ko	3.4 h	12.11	2.9 h	11.75	25.60

Tableau B : Double autojointure (`where t1=t2 and t2=t3`)

Fig. 5.12 – Différentes expérimentations sur les jointures

	Taille	flt_{CQL}	$\mathbb{C}\text{QL}_X$	$\mathbb{C}\text{QL}_P$	$\mathbb{C}$QL	QIZX	QEXO
Q1	1.5 Mb	0.15	0.15	0.15	0.15	0.57	0.74
Q1	29 Mb	2.57	2.58	2.58	2.58	2.16	2.58
Q1	72 Mb	6.61	6.65	6.64	6.62	4.42	5.08
Q1	145 Mb	14.10	14.18	14.15	14.13	8.16	9.31
Q8	1.5 Mb	0.15	0.21	0.21	0.17	1.00	34.51
Q8	29 Mb	2.57	26.03	22.96	13.09	75.90	¿1h
Q8	72 Mb	6.61	156	133	72.81	476	
Q8	145 Mb	14.19	630	542	285	1838	
Q12	1.5 Mb	0.16	0.21	0.21	0.20	0.87	
Q12	29 Mb	2.59	21.22	20.57	14.70	38.30	
Q12	72 Mb	6.68	127	122	86.35	216	
Q12	145 Mb	14.36	481	457	319	824	
Q16	1.5 Mb	0.15	0.16	0.16	0.16	0.62	0.78
Q16	29 Mb	2.57	2.65	2.64	2.63	2.15	2.63
Q16	72 Mb	6.63	6.87	6.85	6.82	4.42	5.08
Q16	145 Mb	14.24	14.60	14.54	14.50	8.16	9.31

(flt = file load time)

FIG. 5.13 – Résumé des résultats des tests XMARK

lise un analyseur lexical externe qui est EXPAT[exp] (mais l'interpréteur $\mathbb{C}$Duce a une option spécifique pour changer utiliser un autre analyseur lexical, PXP[pxp], moins efficace, mais plus agréable pour déboguer), alors que QEXO et QIZX, utilisent des parseurs propres qui leur permettent dans certains cas de ne pas explorer des parties inutiles des documents à interroger. On pourra se référer aux travaux [BCL+05, MS03, BCCN06] qui donnent une approche formelle et des algorithmes qui permettent de n'explorer que les parties du document sur lesquelles une requête est amenée à explorer, et d'améliorer ainsi les temps d'analyse lexicale et d'exécution. Mais dans le cas de QIZX et de QEXO, il s'agit d'algorithmes simples, qui expliquent qu'ils soient meilleurs que $\mathbb{C}$QL. Ce type d'optimisations serait intéressant à rajouter à l'avenir dans $\mathbb{C}$QL. Quand les performances sont déterminés par le moteur de requêtes, comme Q8 et Q12, $\mathbb{C}$QL montre alors de meilleurs résultats[5].

Dans la suite nous détaillons chaque requête séparément pour $\mathbb{C}$QL.

5.2.2.1 Requête Q1

La requête Q1 renvoie les noms des items avec l'`id` égal à '`item20748`' enregistrés en Amérique du Nord, et est donnée dans la figure 5.14. Il s'agit d'une projection sur quatre éléments contenant un prédicat d'égalité. Comme en $\mathbb{C}$QL nous ne disposons pas de la possibilité de mettre un prédicat dans une projec-

[5]Les tests sur QIZX ont été lancés avec la désactivation de l'optimisation sur les jointures.

XQUERY :

```
for $b in document("auction.xml")/site/regions/namerica/item[@id="item20748"]
return $b/name/text()
```

$\mathbb{C}$QL$_X$:

```
select [b]/Xname/Char
from b in [doc]/Xregions/Xnamerica/Xitem
where [b]/@id = ["item20748"]
```

$\mathbb{C}$QL$_P$:

```
select x1
from <_ ..>[(x3::Xregions|_)*] in [doc],
     <_ ..>[(x4::Xnamerica|_)*] in x3,
     <_ id="item20748">_&<.. _>[(x5::Xitem|_)*] in x4,
     <_ ..>[(x2::Xname|_)*] in x5,
     <_ ..>[(x1::Char|_)*] in x2
```

$\mathbb{C}$QL :

```
select s
from <site>[r&Xregions ;_ ] in [doc],
     <regions>[_* n&Xnamerica ;_] in [r],
     <namerica>[(i::Xitem|_)*] in [n],
     <item id="item20748">[ _* <name>s ;_] in i
```

FIG. 5.14 – Requête XMark Q1 - Renvoyer les noms des items avec l'id 'item20748' enregistrés en Amérique du Nord.

tion, nous avons rajouté son équivalent dans la clause `where`. Mais on pourrait quand même, en jouant sur les types pouvoir exprimer la même chose, avec ce type de projection :`[doc]/Xregions/Xnamerica/<item id="item20748">[Any*]`, mais nous avons préféré utiliser la clause `where`, car tous les prédicats d'une expression XPATH ne sont pas exprimables par les types. Mais dans notre cas, la mise en motifs de la condition permet de capturer ce prédicat en motif. On peut voir que le temps mis pour charger le fichier prend une très grande partie du temps. On remarquera pour le document de 145 Mo que le gain est plus visible pour $\mathbb{C}$QL$_P$ et $\mathbb{C}$QL. On voit que la traduction a permis de capturer deux éléments ensemble (l'`id` et le `Xitem`), ce qui explique ce gain. De même pour $\mathbb{C}$QL qui capture l'équivalent de trois motifs en un. La connaissance du type particulier et le fait que chaque item n'ait qu'un nom permet l'écriture d'un tel motif.

XQuery :

```
for $p in document("auction.xml")/site/people/person
let $a := (for $t in document("auction.xml")/site
                                          /closed_auctions/closed_auction
           where ($t/buyer/@person = $p/@id)
           return $t )
return <item person="{$p/name/text()}"> {count ($a)} </item>
```

$\mathbb{C}$QL$_X$:

```
select <item person=([p]/Xname/Char)>
         [ (count ( select t
                    from t in [doc]/Xclosed_auctions/Xclosed_auction
                    where [t]/Xbuyer/@person = ([p]/@id) ) )]
from p in [doc]/Xpeople/Xperson
```

$\mathbb{C}$QL$_P$:

```
select <item person=x1>
         [( count (select t
                   from <_ ..>[(x5::Xclosed_auctions|_)*] in [doc],
                        <_ ..>[(x6::Xclosed_auction|_)*] in x5,
                        t&<_ ..>[(x8::Xbuyer|_)*] in x6,
                        <_ person=x7>_ in x8,
                        <_ id=x9>_ in [p]
                   where [x7] = [x9] ))]
from <_ ..>[(x3::Xpeople|_)*] in [doc],
     <_ ..>[(x4::Xperson|_)*] in x3,
     p&<_ ..>[(x2::Xname|_)*] in x4,
     <_ ..>[(x1::Char|_)*] in x2
```

$\mathbb{C}$QL :

```
select <item person=n>
      [(count(select t
              from <site>[_* <closed_auctions>[t::Xclosed_auction*]]  in [doc],
                   <closed_auction>[_ <buyer person=i2>[ ] ;_] in t
              where i=i2 ))]
from <site>[_* <people>[p::Xperson*] ;_ ] in [doc],
     <person id=i>[<name>n ;_] in p
```

Fig. 5.15 – Requête XMark Q8 - Donner la liste des noms des personnes et le nombre d'items qu'ils ont achetés.

5.2.2.2 Requêtes Q8, Q12 et Q16

La requête Q8 donne la liste des noms des personnes et le nombre d'items qu'ils ont achetés, et est présentée dans la figure 5.15. Elle est assez équivalente dans sa

forme avec la requête Q12 qui donne la liste pour chaque personne, du nombre d'items qu'ils ont en vente dont le prix ne dépasse pas 0.02% de leur revenu, et qui est présentée dans la figure 5.16.

Il est nécessaire de faire intervenir une opération de comptage qui a été écrite en tant que fonction en préambule de la requête et n'apparaît pas ici. Il s'agit d'une requête imbriquée. Il est remarquable de noter que les deux requêtes écrites avec des projections semblent bien plus compactes que leur traduction $\mathbb{C}\text{QL}_P$, et que pourtant les temps d'évaluation de $\mathbb{C}\text{QL}_P$ sont plus rapide. Ce qui suggère là encore que l'utilisation de motifs ayant la même sémantique que des projections fait gagner du temps.

L'utilisation de motifs complexes en ayant la connaissance du type, permet pour un utilisateur averti d'écrire des requêtes compactes, ce qui conduit à un gain de temps considérable ; car par exemple dans la figure 5.15 en quatre motifs, on peut capturer l'équivalent de neuf motifs, et donc de neuf projections.

On peut également faire le même genre de remarque avec la requête Q16 qui renvoie les id des vendeurs d'enchères fermées qui ont au moins un mot-clef en emphase, et qui est donnée dans la figure 5.17. Un point intéressant dans l'expressivité de $\mathbb{C}$QL est montré dans cette requête. L'utilisation de la fonction `not(empty(...))` est illustrée ici avec l'utilisation d'une projection dans la requête $\mathbb{C}$QL. Le résultat de la deuxième projection est capturée dans la variable b qui ne sert pourtant pas dans la suite de la requête, ni dans la construction du résultat. Si b ne capture aucune variable alors aucun résultat ne sera renvoyé, et permet ainsi de simuler la fonction not(empty(...)) de XQuery. Et on retrouve cette opération dans la traduction de la requête, et dans la version $\mathbb{C}$QL. Ce qui signifie que chaque variable utilisée a un sens et nous servira dans la construction de requêtes dans l'interface graphique que nous présenterons au chapitre suivant.

En conclusion, nous montrons dans cette partie sur les jeux de tests XMark, que la traduction améliore les résultats, et permet ainsi une optimisation conséquente. Nous montrons aussi qu'une requête écrite en connaissant les types d'une DTD relativement précise permet d'écrire des motifs précis qui améliorent encore les performances.

5.3 Approche *micro-benchmarks*

Les requêtes précédentes montrent que $\mathbb{C}$QL en tant que langage de requêtes réécrivant ces requêtes en du code $\mathbb{C}$Duce obtient d'honorables résultats. Mais une critique qui peut être formulée sur cette série de tests est que ces tests ont été

XQUERY :

```
for $p in document("auction.xml")/site/people/person
let $l := for $i in document("auction.xml")/site/
                                        open_auctions/open_auction/initial
          where ($p/profile/@income > (5000 * ($i/text())))
          return $i
where $p/profile/@income > 50000
return <items person="{$p/profile/@income}">{count($l)} </items>
```

$\mathbb{C}$QL$_X$:

```
select
<items person=i1>
        [(count( select i2
                 from i2 in [doc]/Xopen_auctions/Xopen_auction/Xinitial
                 where int_of(i1) >> (5000 * int_of([i2]/Char)) ))]
from p in [doc]/Xpeople/Xperson,
     i1 in [p]/Xprofile/@income
where int_of(i1) >> 50000
```

$\mathbb{C}$QL$_P$:

```
select
<items person=i1>
        [( count( select i2
                  from  <_ ..>[(x4::Xopen_auctions|_)*] in [doc],
                        <_ ..>[(x5::Xopen_auction|_)*] in x4,
                        <_ ..>[(x6::Xinitial|_)*] in x5,
                        i2&<_ ..>[(x7::Char|_)*] in x6
                  where int_of(i1) >> (5000 * int_of(x7))))]
from <_ ..>[(x1::Xpeople|_)*] in [doc],
     <_ ..>[(x2::Xperson|_)*] in x1,
     <_ ..>[(x3::Xprofile|_)*] in x2,
     <_ income=i1>_ in x3
where int_of(i1) >> 50000
```

$\mathbb{C}$QL :

```
select <items person=i1 >
      [( count(select i2
               from <site>[_* <open_auctions>[oa::Xopen_auction*] _]  in [doc],
                    <open_auction>[<initial>i2 _*] in oa
               where int_of(i1) >> 5000 * int_of(i2) ))]
from <site>[_* <people>[p::Xperson*] ;_ ] in [doc],
     <person>[ _* pr::Xprofile _* ] in p,
     <profile income=i1>_ in pr
where int_of(i1) >> 50000
```

FIG. 5.16 – Requête XMark Q12 - Pour chaque personne, donner la liste du nombre d'items qu'ils ont en vente dont le prix ne dépasse pas 0.02% de leur revenu.

XQUERY :

```
for $a in document("auction.xml")/site/closed_auctions/closed_auction
where not (empty ($a/annotation/description/parlist/listitem/parlist/
                                        listitem/text/emph/keyword/text()))
return <person id="{$a/seller/@person}" />
```

$\mathbb{C}$QL$_X$:

```
select <person id=(flatten([a]/Xseller/@person))>[]
from a in [doc]/Xclosed_auctions/Xclosed_auction,
     b in [a]/Xannotation/Xdescription/Xparlist/Xlistitem/
                          Xparlist/Xlistitem/Xtext/Xemph/Xkeyword/Char
```

$\mathbb{C}$QL$_P$:

```
select <person id=(flatten([x1]))>[]
from <_ ..>[(x2::Xclosed_auctions|_)*] in [doc],
     <_ ..>[(x3::Xclosed_auction|_)*] in x2,
     a&<_ ..>[(x14::Xseller|_)*]&<_ ..>[(x4::Xannotation|_)*] in x3,
     <_ ..>[(x5::Xdescription|_)*] in x4,
     <_ ..>[(x6::Xparlist|_)*] in x5,
     <_ ..>[(x7::Xlistitem|_)*] in x6,
     <_ ..>[(x8::Xparlist|_)*] in x7,
     <_ ..>[(x9::Xlistitem|_)*] in x8,
     <_ ..>[(x10::Xtext|_)*] in x9,
     <_ ..>[(x11::Xemph|_)*] in x10,
     <_ ..>[(x12::Xkeyword|_)*] in x11,
     <_ ..>[(x13::Char|_)*] in x12,
     b in x13,
     <_ person=x1>_ in x14
```

$\mathbb{C}$QL :

```
select <person id=p>[]
from <site>[ _* <closed_auctions>[ a::Xclosed_auction*]] in [doc],
     <closed_auction>[<seller person=p>_ _* <annotation>
                    [_ <description>[<parlist>[x::Xlistitem* ]] _]] in a,
     <listitem>[(x2::Xparlist|_)*] in x,
     <parlist>[x3::Xlistitem*] in x2,
     <listitem>[(x4::Xtext|_)*] in x3,
     <text>[(x5::Xemph|_)*]   in x4,
     <emph>[(x6::Xkeyword|_)*]  in x5,
     <keyword>[(x7::Char|_)*]  in x6,
     x8 in x7
```

FIG. 5.17 – Requête XMark Q16 - Renvoyer les id des vendeurs d'enchères fermées qui ont au moins un mot-clefs en emphase.

faits sur des requêtes qui comportent un certain nombre de composantes. Et que finalement chaque aspect n'est pas étudié séparément. Il est possible que des motifs soient très bons par exemple sur un type d'arbre, et très mauvais sur d'autres en

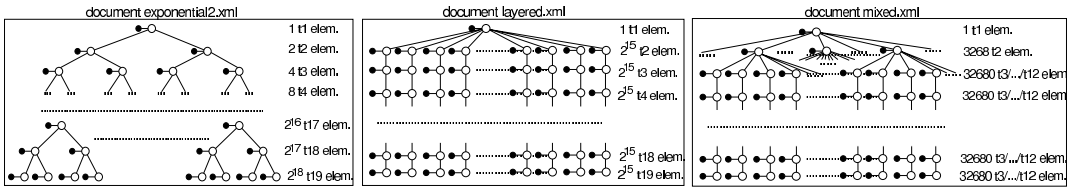


FIG. 5.18 – Squelettes des documents **exponential2.xml**, **layered.xml** et **mixed.xml** utilisés dans les mesures.

fonction de la taille ou de la profondeur. Un travail [MMM06] a été fait dans le cadre de cette thèse pour comparer différents SGBD implantant XQUERY, et $\mathbb{C}$QL, afin de les comparer entre eux, et identifier leurs atouts et leurs faiblesses dans l'exécution de requêtes XQUERY.

Il existe deux types d'approche pour comparer des systèmes en utilisant des benchmarks.

- le premier est *l'application de benchmarks* qui est la voie que nous avons suivie dans la section précédente, en se servant des jeux de tests des « *XML Query Use Cases* » et de XMARK. Il existe aussi XMACH-1 [BR01], X007 [BDL+01] and XBENCH [YÖK04]. Ces systèmes sont utilisés pour évaluer les performances générales en testant le maximum de fonctionnalités du langage de requêtes, en utilisant un jeu de tests fini.
- le second est l'approche *micro-benchmarks* utilisée par MEMBER[AMM05b, AMM05a] et XCHECK[AFMZ06, AM05] qui a été conçue pour vérifier les performances de fonctionnalités isolées d'un système.

5.3.1 Présentation des requêtes

Nous avons considéré l'approche micro-benchmarks, et étudié ces différentes constructions XPATH et XQUERY.

- La navigation XPATH : les expressions étudiées sont de type $t_1/\ldots/t_n$ où n varie de 1 à 19. Ces expressions ont été exécutées sur des arbres binaires, où chaque nœud t_n contient un attribut `id` et a deux fils t_{n+1}, sur les documents *exponential2.xml* et *layered.xml* dont la structure est présentée dans la figure 5.18.
- Les prédicats XPATH : dans ces expressions on demande l'attribut `id` à la position 1 ou `last()` sur la projection précédente. On teste la présence de la descendance jusqu'à n pour des requêtes de type $t_1[t_2/\ldots/t_n]/@\texttt{id}$.
- Les FLWR de XQUERY : Nous avons considéré des requêtes imbriquées avec autant d'itérateur que la profondeur n.
- La construction de nœuds XQUERY : qui consiste à reconstruire de nouveaux

éléments (comme la chaîne de caractères concaténée avec le contenu de tous les `id` au niveau n) ou à copier de nouveaux éléments.

Toutes les requêtes et descriptions des moteurs de requêtes utilisés peuvent être trouvées à cette adresse [AMMM06]. Les moteurs de requêtes qui ont été considérés sont eXist[exi], Galax[Bel], MonetDB[mon] qui possèdent un stockage de données, et Qizx[Fra], Saxon[sax] et $\mathbb{C}$QL. Pour ce qui concerne $\mathbb{C}$QL, comme aucune des requêtes étudiées ne possèdent la navigation descendante, elles peuvent être exprimées aisément en $\mathbb{C}$QL, et ont été optimisées vers le filtrage par motifs en utilisant la traduction $\mathcal{T}$, sauf pour les expressions de chemins utilisant la projection sur descendants qui a été laissée identique. Il existe quelques fonctionnalités d'XQuery que $\mathbb{C}$QL ne possède pas, comme par exemple, l'identification de nœuds. Comme $\mathbb{C}$Duce est un langage fonctionnel basé sur des valeurs, il ne permet pas de distinguer deux nœuds ayant la même valeur sérialisée, et en particulier, $\mathbb{C}$Duce est incapable de tester si deux variables correspondent au même nœud. Le but de ces tests de performance n'est pas de détecter quel système est le meilleur mais de guider les développeurs d'application XQuery (et de langages de requêtes pour XML) pour l'amélioration de leur système, et d'aider éventuellement un utilisateur ne disposant que de requêtes particulières, et lui proposer ainsi le système le mieux adapté à ses besoins.

Nous détaillons chaque requête, et donnons dans la suite les graphiques des temps d'évaluation pour chaque série de requêtes. En ordonnées de chaque graphique, on retrouve le temps d'évaluation qui est la somme du temps de compilation de la requête, d'exécution, et du temps mis pour sérialiser la réponse[6], et en abscisses n, correspondant à la profondeur de la requête, ou de l'expression de chemin : on s'attend intuitivement à ce que les courbes soient croissantes suivant n.

5.3.1.1 Partie XPath

Nous avons mesuré sur le document **exponential2.xml** sept requêtes différentes utilisant des projections XPath, représentées par $\mathbf{Q1.1}(n)$, $\mathbf{Q1.2}(n)$, ..., $\mathbf{Q1.7}(n)$, avec $1 \leq n \leq 19$:

- $\mathbf{Q1.1}(n)$: `/t1/t2/.../t`n

  Cette requête recherche des nœuds avec une profondeur croissante dans le document. La taille de la réponse, et donc des données à sérialiser, de cette requête diminue avec la profondeur de la recherche dans le document. Cette requête a été conçue pour tester la capacité des moteurs de requêtes à ma-

[6] La sérialisation est l'enregistrement du résultat de la requête sur le disque.

nipuler correctement les tailles croissantes des expressions de chemins. Nous avons aussi mesuré $\mathbf{Q1.1}(n)$ sur le document **layered.xml**, qui fournit des résultats intéressants comparés au document **exponential2.xml**.

- $\mathbf{Q1.2}(n)$: `/t1/t2/.../tn/@id`
  Pour différencier l'impact de la navigation de l'impact de sérialisation de sous-arbres dans le résultat, nous avons utilisé la requête $\mathbf{Q1.2}(n)$ qui navigue à la même profondeur que $\mathbf{Q1.1}(n)$ mais qui ne retourne que les attributs `id`, ce qui diminue fortement la taille du résultat.
- $\mathbf{Q1.3}(n)$: `(/t1/t2/.../tn)[1]/@id`
  Cette requête est utilisée pour voir quel moteur de requêtes est capable de prendre avantage du prédicat [1] pour minimiser le temps d'exécution, et de ne pas explorer les autres nœuds au même niveau.
- $\mathbf{Q1.4}(n)$:`(/t1/t2/.../tn)[position()=last()]/@id`
  Cette requête est proche de la requête $\mathbf{Q1.3}(n)$, mais elle utilise le prédicat [position()=last()] qui n'est pas aussi facilement optimisable que le prédicat [1]. Mesurer ces deux requêtes séparément permet de voir quelles techniques d'optimisation sont supportées par le moteur de requêtes.
- $\mathbf{Q1.5}(n)$: `/t1[t2/.../tn]/@id`
  La requête $\mathbf{Q1.5}(n)$ vise à quantifier l'impact d'une navigation existentielle profonde le long des branches. Cette requête vérifie si le moteur de requêtes est capable de ne pas explorer, dès qu'une balise t_i a été trouvée, l'ensemble de ses frères, et ainsi ne pas explorer ses sous-arbres.
- $\mathbf{Q1.6}(n)$: `/t1[t2/.../tn]/t2/.../tn/@id`
  La requête $\mathbf{Q1.6}(n)$ présente une possibilité d'optimisation évidente (car la branche existentielle, peut être supprimée sans changer la sémantique de la requête) ; ses résultats sont donc à comparer avec la requête $\mathbf{Q1.2}(n)$.
- $\mathbf{Q1.7}(n)$: `//tn`
  Et en dernier la requête $\mathbf{Q1.7}(n)$ recherche tous les éléments d'une balise donnée. Ses résultats sont à comparer avec ceux de la requête $\mathbf{Q1.1}(n)$, pour voir si la connaissance qu'a un utilisateur du chemin à parcourir pour capturer les éléments désirés, simplifie ou non la tâche du moteur de requêtes. Dans le cas de $\mathbb{C}$QL, on a juste utilisé l'opérateur //, sans l'optimisation de traduction $\mathcal{T}$ en motifs.

5.3.1.2 Partie XQuery

Nous avons mesuré sur le document *mixed.xml* un ensemble de six requêtes XQuery représentées par **Q2.1**(n), **Q2.2**(n), ..., **Q2.6**(n), où $0 \leq n \leq 9$, telles que définies ci-après :

- **Q2.1**(n) :

```
for $x in /t1/t2
return <res>{ for $y in $x/*/*/.../*
                return fn :data($y/@id)}
       </res>
```

  où l'expression de chemin est liée à **$y** et comporte n étapes de chemins sur les fils à partir de la variable **$x**. Cette requête va tester la performance de la navigation dont la profondeur augmente dans la clause `return`, tandis que la taille des résultats reste constante et petite.

- **Q2.2**(n) :

```
for $x in /t1/t2
return <res>{$x/*/*/.../*}</res>
```

  où l'expression de chemin dans la clause `return` comporte n étapes de chemins sur les fils à partir de **$x**. La requête **Q2.2**(n) mélange une navigation dont la profondeur augmente, avec un nombre de sous-arbres à copier dans le résultat qui décroît.

- **Q2.3**(n) :

```
for $x in /t1/t2
return <res>{$x//t3, $x//t4, ...,
                        $x//t(n + 2)}</res>
```

  La requête **Q2.3**(n) a été conçue pour renvoyer des résultats dont la taille est supposée augmenter avec n, compte tenu du fait que les éléments **t3**, **t4**, ..., **t13** sont uniformément distribués dans les niveaux trois à treize du document **mixed.xml**. De plus, la nature de la requête fait que tous les éléments à rechercher sont dispersés dans le document. Cela nous permet de vérifier comment le moteur de requêtes gère des résultats d'une taille conséquente et dispersés.

- **Q2.4**(n) :

```
for $x in /t1/t2
return <res>{$x/*[position()≤ n]}</res>
```

  La requête **Q2.4**(n) renvoient des résultats dont la taille croît linéairement avec n, cependant dans ce cas, les éléments à renvoyer sont préalablement

groupés dans le document d'origine dans des sous-arbres contigus. La performance de Q2.3(n) comparés avec ceux de **Q2.4**(n) devrait nous fournir d'intéressantes indications sur la stratégie (éventuelle) de clustering des nœuds, utilisée par le système.

- **Q2.5**(n) :

```
for $x in /t1/t2
return $x/*[position()≤ n]
```

La requête **Q2.5**(n) est similaire à la requête **Q2.4**(n), sauf que **Q2.5**(n) ne construit pas de nouveaux éléments. Cette requête **Q2.5**(n) peut être écrite en XPath mais elle a été gardée dans cette série de requêtes pour être comparée avec **Q2.4**(n). Comme **Q2.5**(n) ne construit pas de nouvel élément, il est possible que son évaluation soit plus rapide que **Q2.4**(n) car aucune copie d'arbre n'est nécessaire.

- La requête **Q2.6**(n) a un nombre d'imbrications de clauses `return` qui augmente . Cette requête recherche les éléments **t13** : voici quelques exemples de cette requête pour $n = 0$, $n = 1$ et $n = 2$.

Q2.6(0) :

```
for $x in /t1/t2 return
<res>{for $x1 in $x/* return
        <res>{$x1//t13}</res>}
</res>
```

Q2.6(1) :

```
for $x in /t1/t2 return
<res>{for $x1 in $x/* return
        <res>{for $x2 in $x1/* return
                <res>{$x2//t13}</res>}
        </res>}
</res>
```

Q2.6(2) :

```
for $x in /t1/t2 return
<res>{for $x1 in $x/* return
        <res>{for $x2 in $x1/* return
               <res>{for $x3 in $x1/* return
                       <res>{$x3//t13}</res>}
               </res>}
        </res>}
</res>
```

La requête $\mathbf{Q2.6}(n)$ renvoie des sous-arbres contenant un nombre $n+2$ d'imbrications d'éléments <res>. Sur le document **mixed.xml**, cette requête retournera 3268 éléments <res> qui correspondent au nombre d'éléments **t2** dans ce document. Chacun de ces éléments <res> contiendra 10 copies des éléments **t13**. Comme n croît, cependant le nombre de couches d'éléments <res> dans lesquels les éléments **t13** sont contenus augmentera. Cette requête et son document ont été choisis pour détecter seulement le coût de l'imbrication de résultats.

Toutes ces requêtes ont été réalisées sur différents ordinateurs et sur différents systèmes de requêtes. Nous détaillons dans la suite les tests et les résultats des requêtes[7] pour ce qui concerne $\mathbb{C}$QL.

5.3.2 Résultats de $\mathbb{C}$QL

Ces tests ont été exécutés sur un ordinateur PENTIUM 4 avec 3 GHz , 512 Mo de RAM, sur LINUX DEBIAN 2.6.16.

5.3.2.1 Partie XPATH

Les résultats pour les requêtes de cette partie sont détaillés dans la figure 5.19. Les temps d'exécution de $\mathbf{Q1.1}(n)$ et $\mathbf{Q1.7}(n)$ sont les plus importants et décroissent en fonction de n, dû à la diminution de la taille du résultat de la requête. Un des problèmes qui a été soulevé, est que la requête $\mathbf{Q1.7}(n)$ pour $n = 19$, a provoqué un débordement de la pile ("*stak overflow*"). Ce débordement est provoqué par l'implantation de la projection sur descendants, en une pile sur une fonction récursive, qui capture tous les éléments filtrés, pour les renvoyer à la

[7] On retrouvera le code des requêtes à l'adresse[AMMM06].

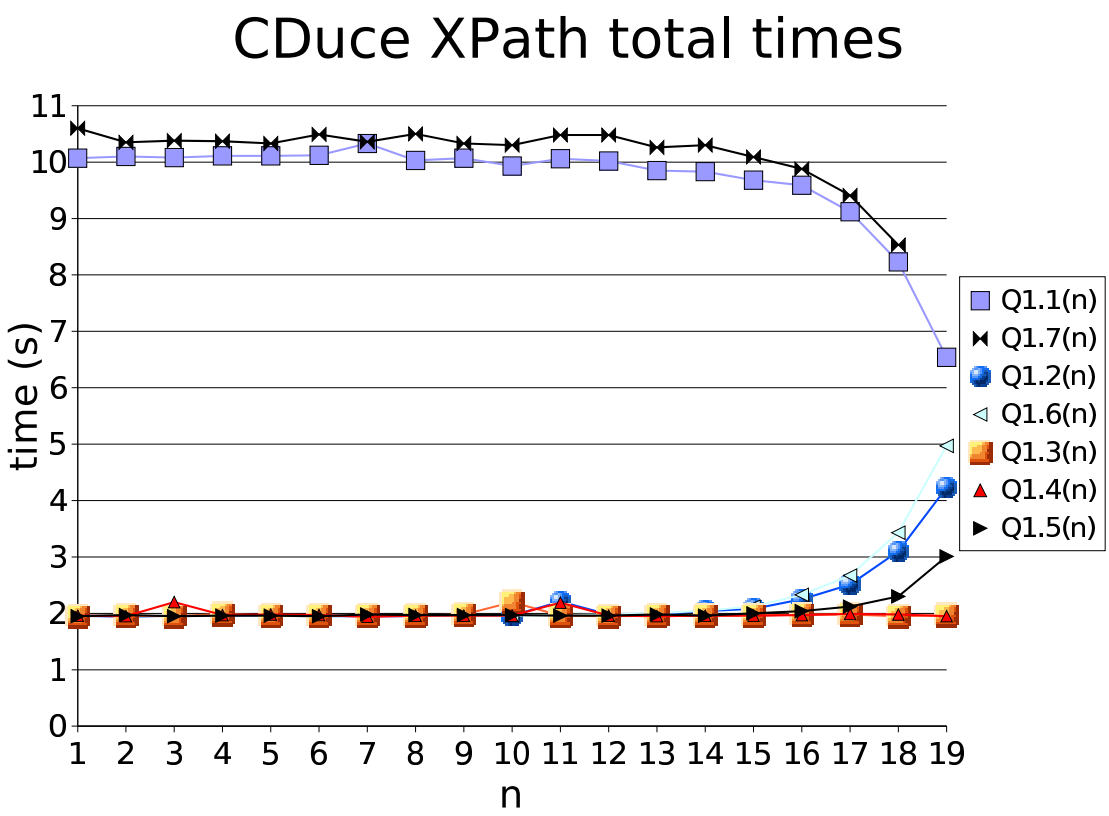


FIG. 5.19 – Temps complets pour la partie XPATH sur le document *exponential2.xml*.

fin de l'exécution. Cependant ces courbes sont très similaires, bien que légèrement plus importants pour $\mathbf{Q1.7}(n)$.

Les requêtes $\mathbf{Q1.3}(n)$ et $\mathbf{Q1.4}(n)$ ont un temps d'évaluation très court et constant. La traduction de ces requêtes XPATH en $\mathbb{C}$QL, combinée avec le filtrage par motifs de $\mathbb{C}$Duce, permet de capturer aisément les prédicats [1] et [position()=last()].

Le temps de $\mathbf{Q1.2}(n)$ montrent une croissance exponentielle car le nombre de nœuds retournés croît exponentiellement. La même observation tient pour $\mathbf{Q1.6}(n)$, avec une augmentation plus importante que $\mathbf{Q1.2}(n)$, bien qu'elles soient équivalentes. La raison est que la traduction génère deux fois plus de motifs que pour $\mathbf{Q1.2}(n)$.

Le temps de $\mathbf{Q1.5}(n)$ croît sensiblement pour les dernières valeurs de n. La raison est que le nombre de motifs augmente avec n, et les courbes reflètent la complexité d'évaluation basée sur les automates d'arbres pour de tels motifs.

Les temps reportés sont la somme de

– du temps de construction des automates (la compilation), déterminée par la requête, et de la DTD

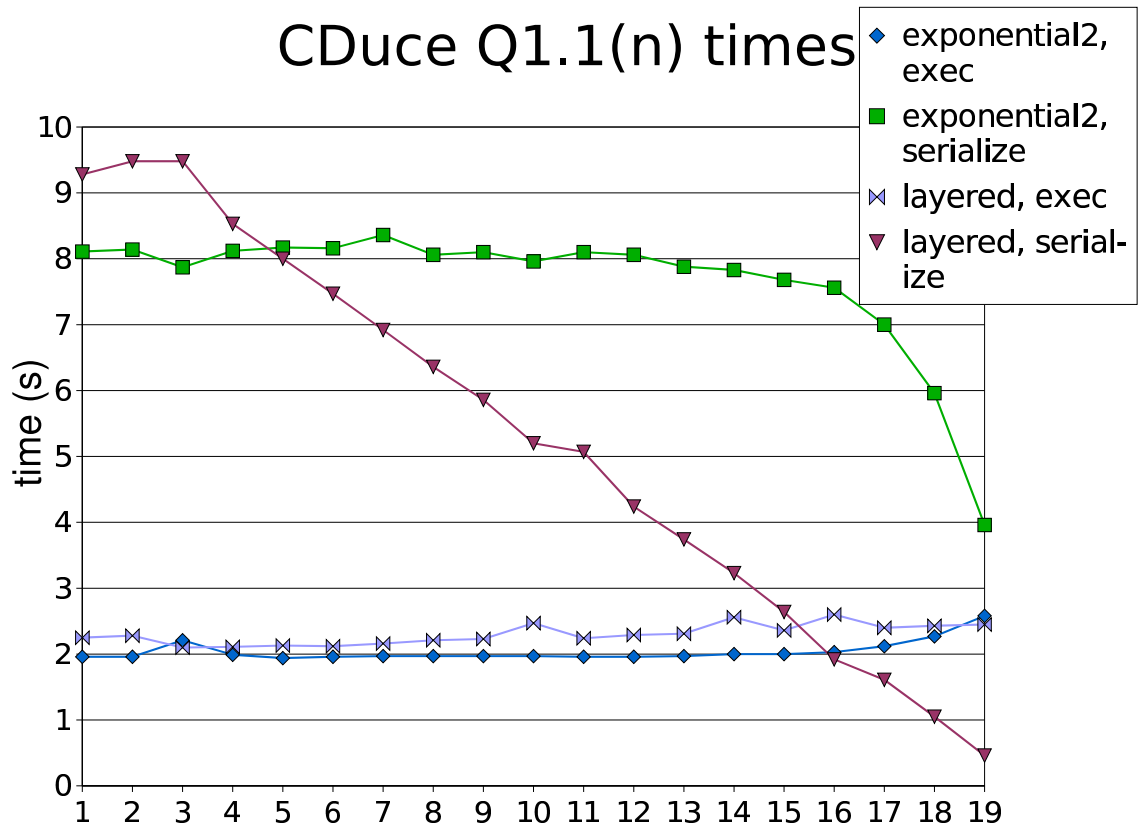


FIG. 5.20 – Différents temps pour la requête $\mathbf{Q1.1}(n)$ sur différents documents.

- **du temps d'exécution**, pendant lequel les résultats sont construits mais non sérialisés
- et **du temps de sérialisation**, qui correspond à l'enregistrement d'un fichier XML contenant le résultat de la requête.

Nous avons raffiné l'analyse des temps d'évaluation de $\mathbb{C}$QL dans la figure 5.20, qui montre uniquement le temps d'exécution ainsi que le temps de sérialisation pour la requête $\mathbf{Q1.1}(n)$ sur les documents **exponential2.xml** et **layered.xml**.

Le temps de sérialisation décroît en fonction de la taille du résultat et du nombre de sous-arbres retournés. Le temps d'exécution est presque constant bien que n augmente, ceci est dû à l'efficacité du filtrage par motifs.

Dans les résultats ci-dessus nous n'avons pas pris en compte les informations de types de la DTD structurant précisément les documents. Nous avons relancé les tests avec la prise en compte de la DTD. Les résultats sont montrés dans la figure 5.21. Pour les requêtes $\mathbf{Q1.1}(n)$ à $\mathbf{Q1.7}(n)$, les temps d'évaluation sont presque équivalents avec ceux de la figure 5.19, avec un petit gain d'efficacité grâce à la DTD, pour les requêtes $\mathbf{Q1.5}(n)$ et $\mathbf{Q1.6}(n)$.

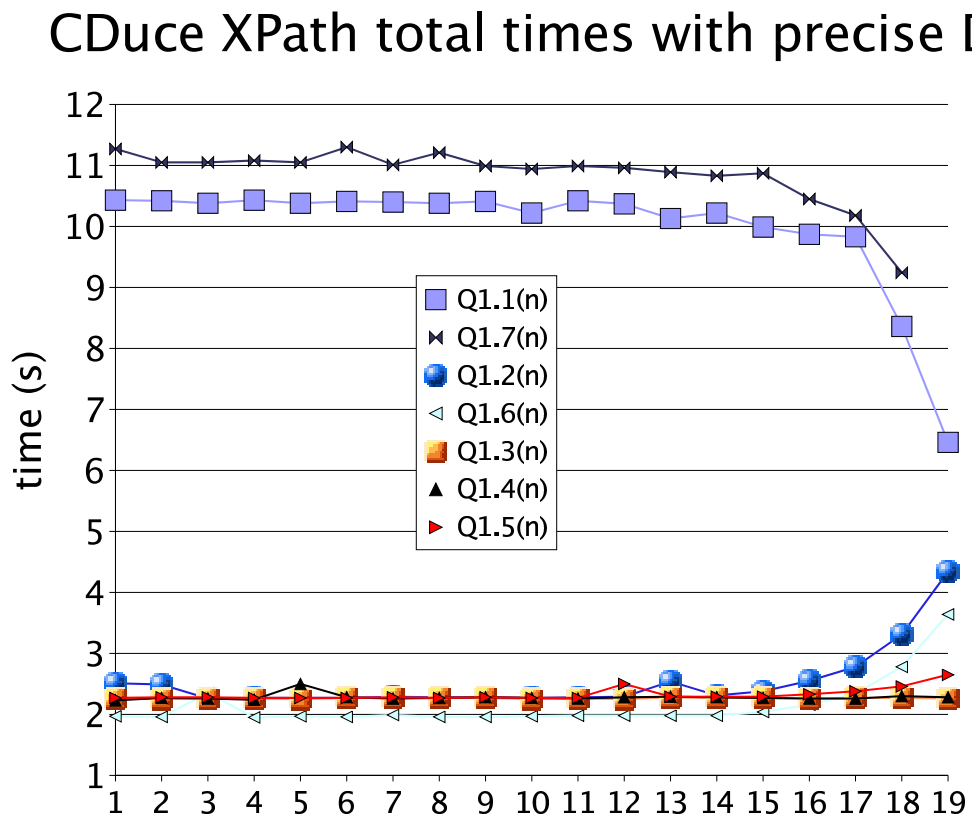


FIG. 5.21 – Temps complets pour la partie XPATH sur le document *exponential2.xml* en présence d'une DTD précise

5.3.2.2 Partie XQUERY

La figure 5.22 présente les temps d'évaluation de ℂQL sur les requêtes de la partie XQUERY. La requête $\mathbf{Q2.1}(n)$ augmente très légèrement avec n, qui reflète une augmentation mince du temps mis par la navigation et par une légère sérialisation. Le temps d'évaluation de la requête $\mathbf{Q2.2}(n)$ décroît comme la taille des sous-arbres retournés dans le résultat. La requête $\mathbf{Q2.3}(n)$ qui construit un résultat d'une taille importante échoue pour $n \geq 2$. La raison est que chaque élément à sérialiser en ℂDuce est écrit dans une chaîne de caractères OCAML, dont la taille maximum est contrainte par ce langage. Les éléments renvoyés par la requête $\mathbf{Q2.3}(n)$ augmentent avec n, et s'arrêtent dès qu'il n'y a plus d'espace disponible. C'est un problème qu'il faudra supprimer en étudiant la possibilité d'ajouter une structure de données qui permettent l'écriture dans un tampon intermédiaire. Ce problème a été aussi rencontré par GALAX qui utilise également OCAML.

Les requêtes $\mathbf{Q2.4}(n)$ et $\mathbf{Q2.5}(n)$ ont les mêmes temps d'évaluation, car ℂDuce, ne copie pas les arbres mais en crée juste une nouvelle valeur. Et la requête $\mathbf{Q2.6}(n)$ augmente linéairement avec les niveaux d'imbrication du résultat.

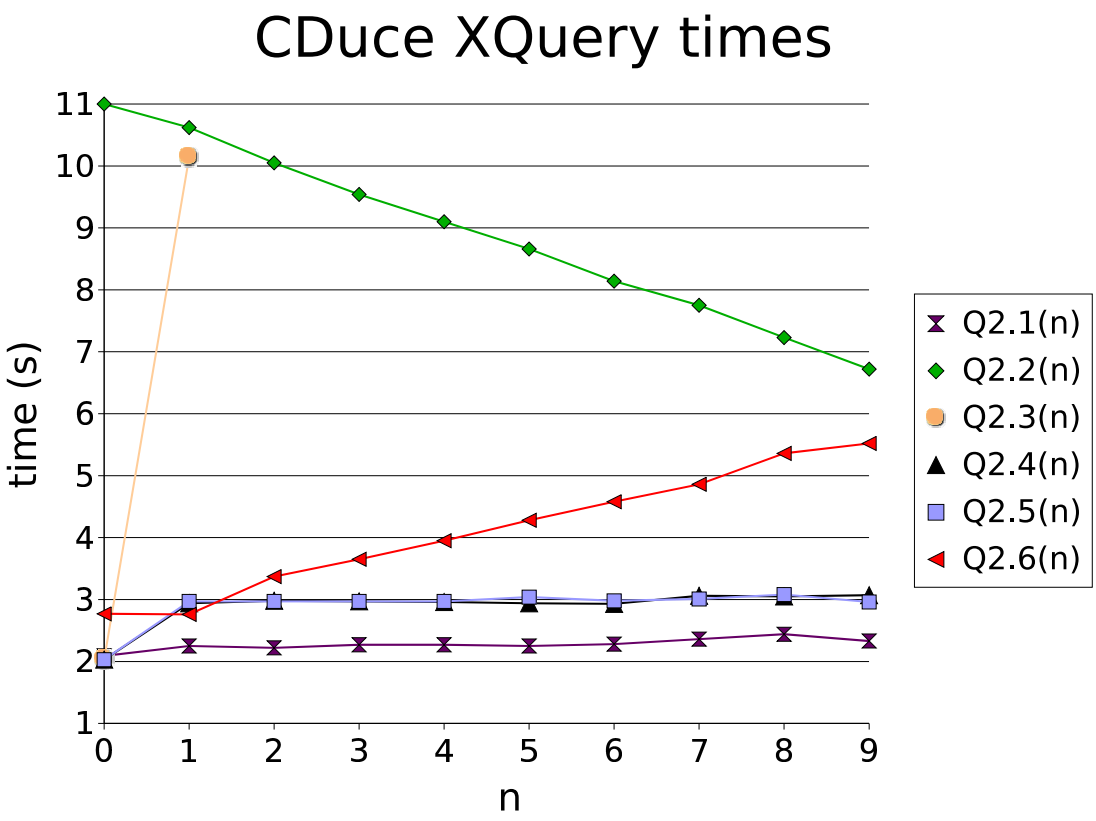


FIG. 5.22 – Temps complets pour la partie XQUERY sur le document *mixed.xml*.

5.3.3 Quelques performances comparées

Dans l'article [MMM06] plusieurs systèmes sont comparés et se comportent différemment suivant les requêtes. Nous ne recopierons pas tous les résultats, nous détaillons juste pour la partie XPATH certains graphiques, pour mettre en valeur une partie de ces résultats.

Ces résultats sont présentés dans la figure 5.23. On remarque tout de suite que ces systèmes ne se comportent pas de la même manière. Par exemple les temps de la requête Q1.1(n) sont quasiment constants dans les systèmes MONETDB, GALAX, EXIST, tandis qu'ils augmentent avec n pour QIZX, et qu'ils décroissent pour SAXON et $\mathbb{C}$QL.

Un cas intéressant est QIZX pour l'opération de projection sur descendants. En comparant les requêtes Q1.1(n) et Q1.7(n), on remarque une évaluation de la projection sur descendants plus longue pour $n \leq 12$ et plus rapide après. Cela semble suggérer que QIZX indexe les balises pour ce genre d'opération, expliquant ces différences de temps, et surtout le gain de temps de 25% pour $n = 19$, qui est le seul système à gagner du temps entre Q1.1(n) et Q1.7(n).

Les requêtes Q1.2(n) et Q1.3(n) qui comparent les optimisations de position sont équivalentes pour tous les systèmes, sauf pour GALAX, qui a besoin

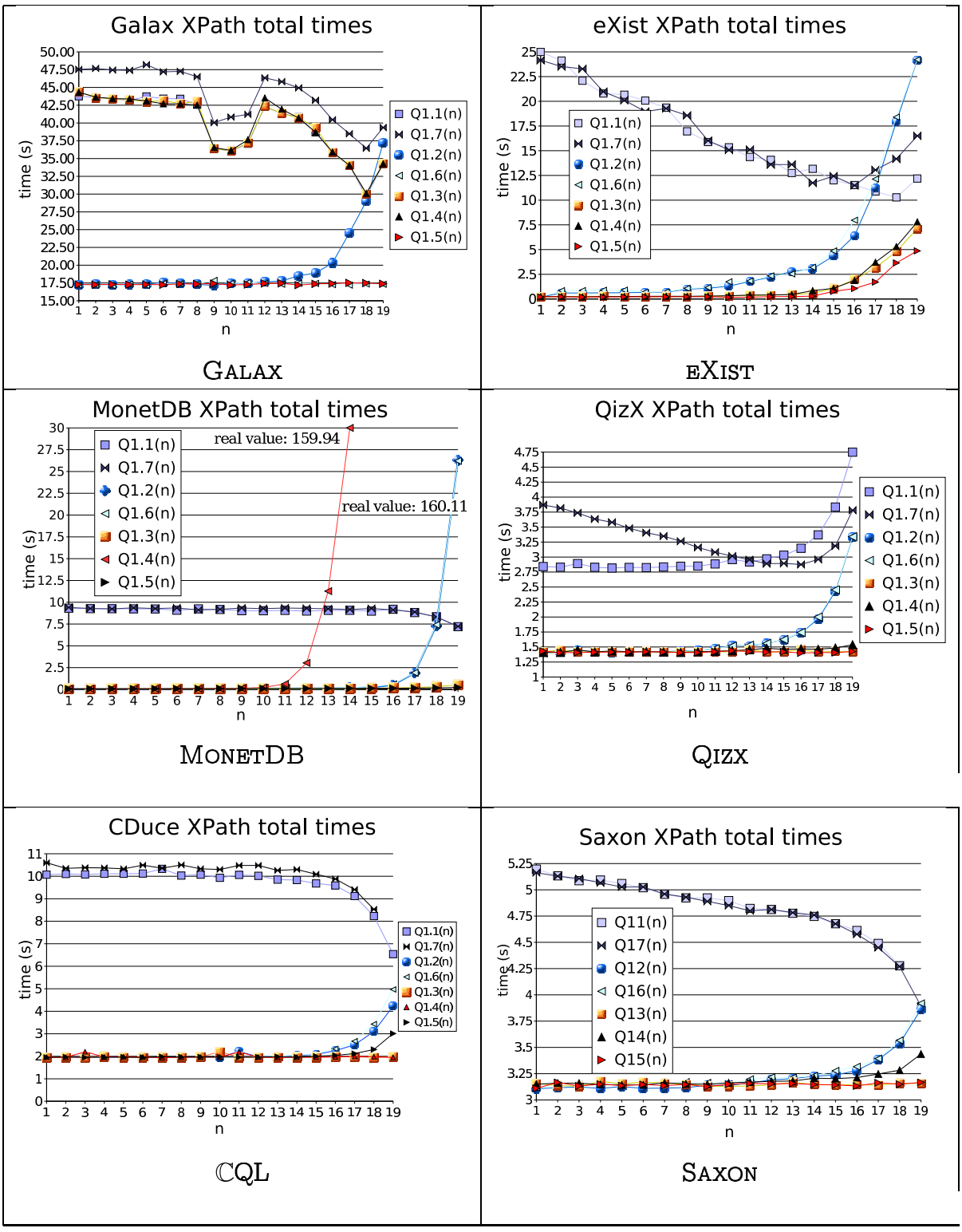


FIG. 5.23 – Tableau comparatif des résultats de la partie XPATH sur le document *exponential2.xml*.

de construire tous les résultats intermédiaires.

On peut comparer aussi les requêtes Q1.2(n) et Q1.6(n) sur les branches existentielles. Ces requêtes sont identiques, et renvoient le même résultat. Dans le cas de EXIST, MONETDB,QIZX et SAXON, les temps sont les mêmes, tandis pour GALAX la différence est relativement grande, et pour CQL il y a une légère différence,

due au nombre de motifs nécessaires. Pour respecter la sémantique de la requête, il y a n motifs pour tester l'existentialité, et n motifs pour la capture pour Q1.6(n), ce qui explique cette différence de temps.

Pour MonetDB, les courbes montrent un comportement très différent pour Q1.2(n), Q1.4(n) et Q1.6(n). En comparant Q1.1(n) et Q1.2(n) on note une différence notable qui semble due à la présence de l'axe attribut dans l'expression de chemin. Pour Galax, on remarque un creux curieux pour $n = 9$, qui serait dû à la sérialisation lors de l'affichage d'OCaml.

5.4 Conclusion

Les tests effectués sur les « *XML Query Use Cases* » confirment que l'utilisation du déconstructeur de filtrage par motifs, apporte de meilleurs résultats de performance que la navigation par chemins. C'est un résultat qui s'explique par le fait que les motifs permettent de capturer plusieurs sous-arbres en même temps (ce qui est le cas pour $\mathbb{C}$QL dans toutes les requêtes précédentes). Le filtrage par motifs est aussi compliqué à écrire pour un utilisateur non averti, au contraire de la navigation par chemins, proche du standard XQuery. L'algorithme de traduction des chemins en motifs, a donc un réel intérêt, dans le cadre de l'optimisation de requêtes par un moteur de requêtes.

La remontée des sélections montre que les techniques de bases de données classiques sont aussi importables dans $\mathbb{C}$QL, et qu'elles permettent de diminuer dans certains cas les temps d'exécution. Nous pourrions aussi rajouter les optimisations sur les jointures qu'utilise Qizx, mais la principale raison pour laquelle nous préférons ne pas implanter ce type d'optimisation dans $\mathbb{C}$QL est que, dans des perspectives orientées bases de données, la génération dynamique d'index, de tables de hashage,...ne doivent pas être faites au niveau du langage de requêtes mais doivent être déléguées à l'optimiseur (physique) de requêtes. En effet, seul l'optimiseur doit être capable de choisir le plan d'optimisation se fondant sur la connaissances des structures de données, des modèles de coûts, et d'ainsi utiliser les algorithmes adéquats. En outre, les optimisations sont souvent dépendantes des données, et ne devraient pas être codées dans le compilateur. Les quelques exemples reportés ici tendent à confirmer cet aspect. Un travail intéressant serait de coupler dans la suite, $\mathbb{C}$QL avec un stockage persistent et d'étudier les optimisations physiques qui peuvent s'y appliquer. En particulier étudier la possibilité de ne pas explorer les parties de documents inutiles pour une requête, ou un jeu de requêtes donné, en s'inspirant des travaux[BCCN06] qui utilisent le typage, pour

des requêtes utilisant XPATH.

Dans l'étude préliminaire[MMM06] utilisant les *microbenchmarks* pour étudier chaque composante de requêtes, il a été mis en lumière :

- la nécessaire modification de l'implantation de la projection sur descendants. Ce point devrait être amélioré grâce au travail de thèse de Kim Nguyễn qui cherche une manière d'approximer le type de l'opérateur de projection sur descendants[Ngu04].
- la nécessité de sérialiser des résultats intermédiaires qui doivent l'être, pour éviter le cas de la requête Q2.3(n). Une méthode résolvant ce point, ainsi que le point précédent, serait d'utiliser un tampon annexe dans le cœur de $\mathbb{C}$Duce.
- que l'absence de l'identité de nœuds que l'on retrouve dans la spécification de XQUERY permet dans certains cas d'être plus rapide, au détriment d'une perte d'expressivité. Cela pourrait être un point intéressant que d'étudier la possibilité de rajouter cette identification de nœuds dans le langage fonctionnel $\mathbb{C}$Duce.
- et enfin le problème du temps mis par un grand nombre de motifs. Ces motifs pourraient être composés en un seul motif.

Ce travail montre donc que les projections réécrites par l'algorithme de traduction $\mathcal{T}$ en motifs permettent de gagner en performance, mais un utilisateur avec une connaissance de la DTD décrivant finement les documents peut écrire des motifs complexes qui permettent de gagner encore plus. Partant du constat qu'il est difficile d'écrire des motifs complexes pour un utilisateur débutant, nous proposons dans la partie suivante une interface graphique qui va offir la possibilité d'écrire de tels motifs en étant guidé à tout instant par la DTD d'un document.

Chapitre 6

Interface graphique : Pattern by Example

Sommaire

6.1 Motivations 123

6.2 Différents langages de requêtes graphiques 124

6.2.1 QBE : un langage graphique pour le relationnel 125

6.2.2 Langages graphiques pour XML 126

6.3 Présentation de PBE 129

6.3.1 Visite guidée 129

6.4 Description formelle de PBE 138

6.4.1 Définitions 139

6.4.2 Syntaxe des tableaux 141

6.4.3 Sémantique 143

6.4.4 Algorithme de traduction d'une requête PBE en une requête ℂQL 144

6.5 Conclusion 158

6.1 Motivations

Dans les sections précédentes nous avons présenté ℂDuce, puis le langage de requêtes ℂQL qui utilise deux types de déconstructeurs. L'interrogation de requêtes ℂQL a des résultats comparables aux implantations XQuery du moment. L'étude des performances des différents types de requêtes ℂQL a montré que dans bien des cas, les requêtes les plus performantes étaient celles qui utilisaient le filtrage par motifs. Bien qu'une requête écrite avec des projections puisse se réécrire avec des motifs, il n'empêche qu'un utilisateur confirmé, et connaissant

le typage précis, via la DTD, des documents XML sur laquelle une requête va s'appliquer pourra dans bien des cas en écrire une plus fine et donc plus efficace. On retrouve de tels cas dans les requêtes des figures 5.3, 5.10 et 5.17 : dans les requêtes $\mathbb{C}$QL et $\mathbb{C}\text{QL}_P$ de la première figure, l'utilisateur sait qu'il n'y a qu'un seul fils `<publisher>` pour un livre, donc il le rajoute directement dans le motif qui capture à partir du livre, le titre et l'année, au lieu de passer par une variable et une nouvelle ligne dans la clause `from`. Dans les requêtes $\mathbb{C}$QL et $\mathbb{C}\text{QL}_P$ de la dernière figure, il sait qu'il existe pour une enchère ouverte, un seul chemin pour capturer la séquence des `<listitem>`, et va donc écrire dans un seul motif complexe, la capture de cette sous-séquence de profondeur quatre, au lieu de passer par quatre motifs différents. Dans ces exemples cette particularité tient à la connaissance de la singularité de certains types ; il existe dans les motifs en général des possibilités de finesse qu'on ne peut pas exprimer par des projections telles que celles définies dans $\mathbb{C}$QL ou XQuery.

Écrire de tels motifs pour un utilisateur débutant, devient vite une gageure. Dans le but d'aider ce programmeur à écrire des requêtes performantes sans avoir connaissance des motifs ni des projections, nous avons défini une interface graphique (PBE : « *Pattern by Example* ») qui tire parti des connaissances sur les expressions régulières des entités de la DTD, et qui se sert du sous-typage de $\mathbb{C}$Duce pour guider l'utilisateur dans l'écriture de requêtes, en lui indiquant par exemple le type de la requête, ou des parties de la requêtes, et pour générer des requêtes complexes. Dans ce chapitre nous ferons un bref état de l'art des différents langages de requêtes graphiques qui existent en insistant sur les langages « QBE » et « XQBE » puis nous ferons une visite guidée de notre outil graphique « PBE », et ensuite nous présenterons la syntaxe et la sémantique formelles, suivies d'un algorithme de construction de requêtes $\mathbb{C}$QL à partir de requêtes PBE.

6.2 Différents langages de requêtes graphiques

Nous présentons ici brièvement QBE qui fut le premier langage de requêtes graphiques, puis XQBE, qui s'inspire de QBE pour XML, et qui génère des requêtes XQuery. PBE est très proche de ces deux langages, empruntant l'utilisation de tableaux d'exemples de QBE, et transposant le cadre des données XML de XQBE à ces tableaux et aux motifs.

6.2.1 QBE : un langage graphique pour le relationnel

« *Query-by-Example* »(QBE) [Zlo75, Zlo77] est un langage de requêtes pour interroger de données relationnelles. Il propose essentiellement une interface graphique qui permet à l'utilisateur d'écrire des requêtes en manipulant des tableaux d'exemples. QBE est relativement facile à comprendre, et a surtout été prévu pour des requêtes qui ne sont pas trop complexes, et qui peuvent être exprimées par de tels tableaux. QBE a été développé à IBM par Moshé Zloof dans les années 1970, et de nombreux produits ont suivi cette approche comme Paradox ou Microsoft Access[Cam92].

Considérons le schéma relationnel suivant qui servira de base pour les exemples de requêtes suivants :

`Emp`(*name* : **string**, *sal* : **integer**, *dept* : **string**)

`Sales`(*dept* : **string**, *item* : **string**)

La relation `Emp` correspond à l'ensemble des employés avec leur nom, leur salaire, et leur département. La relation `Sales` correspond aux produits vendus dans chaque département.

Pour écrire une requête, un utilisateur doit créer des tables d'exemples, et dispose de plusieurs objets pour les remplir. Il y a les variables dont les domaines correspondent au nom de colonne où elles apparaissent et sont soulignées, pour les différencier des constantes. Pour sélectionner un champ, et avoir ainsi le résultat de la requête, il faut utiliser la commande « `P.` » (pour « print »). Ce tableau d'exemple donne l'ensemble des noms des salariés qui travaillent dans le département D1 :

Emp	name	sal	dept
	`P.`		D1

On écrit donc la commande `P.` dans la colonne des noms des employés, et la constante"D1" dans la colonne correspondant au département de la table d'exemples correspondant à la relation Emp. La colonne *sal* est vide car elle n'est pas requise pour écrire cette requête. Cette requête comme toutes celles déduites des tables d'exemples, peut être écrite aisément en calcul à variable domaine, par $\{< N > \mid \exists S, D(< N, S, D1 >) \in Emp\}$.

Les tableaux d'exemples de la figure 6.1 donnent une exemple plus complet de leur utilisation.

Il y a deux tableaux, un pour la relation Emp, et l'autre pour la relation Sales. Cette requête renvoie l'ensemble des employés tels qu'il existe un employé (représenté par la variable "EMP1") qui a un salaire compris entre 10.000 et 15.000, et qui travaille dans un département vendant des stylos. Dans le calcul à

Emp	name	sal	dept
	P.EMP1	>10,000	D1
	P.EMP1	<15,000	D1

SALES	DEPT	ITEM
	D1	PEN

FIG. 6.1 – Une requête QBE : Donner tous les employés dont le salaire est compris entre 10.000 et 15.000, et qui travaillent dans le département qui vend des stylos.

variable domaine, cette requête s'écrit : $\{< EMP1 > | \exists S, D1(< EMP1, S, D1 > \in Emp) \wedge (< D1, \texttt{"PEN"} > \in Sales) \wedge (S > 10,000) \wedge (S < 15,000)\}$. Il peut donc y avoir des jointures si des tableaux ont des variables en commun. On a la possibilité aussi de poser plusieurs contraintes sur une même relation sur des lignes différentes d'un même tableau, ce qui induit la conjonction de ces contraintes.

Avec QBE, on peut exprimer différentes opérations comme la duplication, le tri, les agrégations, la négation, la possibilité de poser des conditions plus complexes dans un objet particulier. Il existe aussi la possibilité de poser des conditions dans une « condition box ».

On trouvera une présentation détaillée dans le chapitre 6 de *Database Management Systems* [RG02] ou dans le chapitre 7 de *Foundations of Databases* [AHV95].

6.2.2 Langages graphiques pour XML

Il existe différents langages graphiques pour des données semi-structurées ou XML. QSByE(Querying Semi-structured data by Example)[FLdS01] est une interface graphique qui étend QBE et utilise des tableaux imbriquées pour manipuler des données semi-stucturées. Miro-web [BCD+99] est basé sur les arbres et implante XML-QL[DFF+98]. EQUIX [CKK+99] est un langage basé sur des formulaires qui sont automatiquement générés à partir de documents XML et de leur DTD. Il existe bien d'autres langages visuels de requêtes comme BBQ[MP00], Pesto[CHMW96], QURSED[PPV05], Xing[Erw03].

Nous ne les présenterons pas, mais parmi tous ces langages graphiques nous ne détaillons que XQBE"*XQuery by Example*"[BCC04, BCC05].

La principale différence entre QBE qui est basé sur l'utilisation de tables relationnelles et XQBE, est que ce dernier est basé sur l'utilisation d'arbres XML pour représenter des requêtes. XQBE a été conçu avec l'objectif d'être intuitif, et d'être directement traduit en XQUERY avec une interface pouvant être utilisée avec un moteur de requêtes XQUERY.

XQBE inclut la majeure partie de la puissance expressive de XPATH (exceptées

quelques fonctions et opérateurs, comme la fonction `position()`). XQBE permet de créer des requêtes imbriquées, de créer de nouveaux éléments,etc.

XQBE peut être vu comme un successeur de XML-GL[CDF01], avec cependant l'ajout de nouvelles fonctionnalités spécifiques à XQUERY. Pour illustrer XQBE, nous montrons une requête, dans la figure 6.2, permettant d'exprimer la requête Q1 des"*XML Query Use Cases*"[CFF+03].

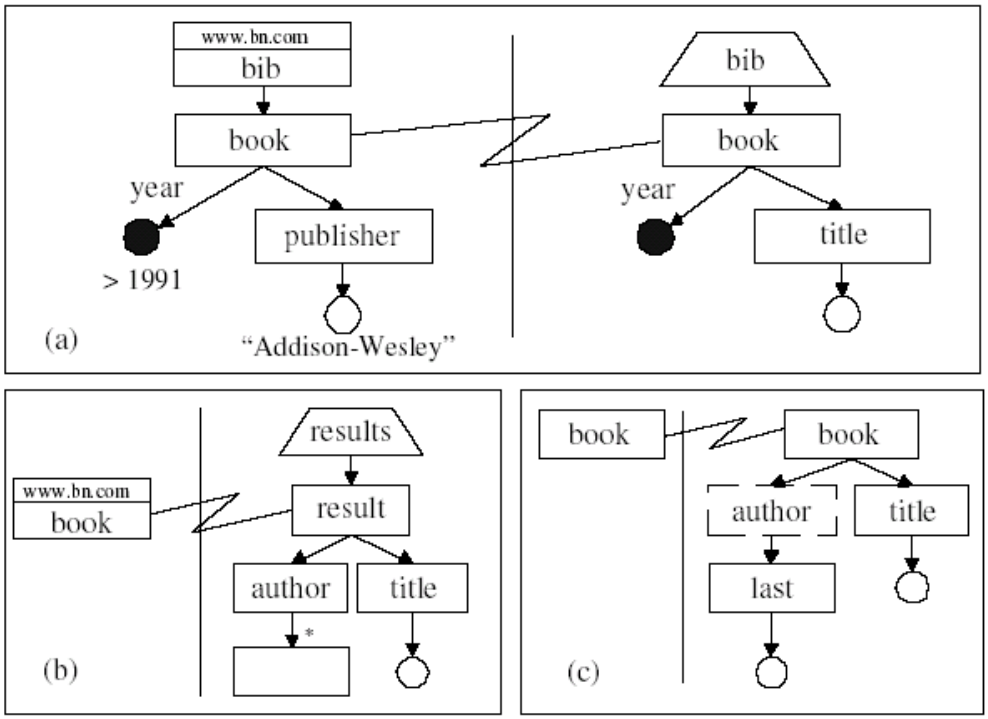


FIG. 6.2 – Requêtes XQBE Q1 (a), Q2 (b) et Q3 (c)

La requête Q1 qui renvoie la liste des livres publiés par Addison-Wesley après 1991, s'écrit en XQUERY :

```
<bib>
{ for $b in document("www.bn.com/bib.xml")/bib/book
  where $b/publisher="Addison-Wesley" and $b/@year>1991
  return <book year="{$b/@year}"> {$b/title} </book> }
</bib>
```

La requête visuelle XQBE permettant de construire Q1 est présentée dans l'image (a) de figure 6.2. Chaque requête dispose d'une ligne séparant la zone source à gauche de la zone de construction à droite. Ces deux zones contiennent des graphes avec des labels qui représentent des fragments de documents XML, et permettent d'exprimer des propriétés sur ces fragments (conditions sur des valeurs, tri,...). La zone source permet de décrire les données XML qui doivent être filtrées pour construire le résultat, et la zone de construction permet de spécifier quelles sont

ces données qui doivent être utilisées, et éventuellement quels nouveaux éléments doivent être créés. Les correspondances entre ces données de la source et du résultat sont exprimées par des relations représentées par des lignes qui vont de la zone source à la zone de construction, et permettent de connecter ainsi les nœuds de la source vers ceux qui seront utilisés dans le document de sortie.

Les éléments XML sont représentés par des rectangles annotés de leur balise, les attributs par des disques noirs avec leur noms sur le trait liant ce cercle au rectangle de leur élément, et les données textuelles par des cercles. Tous les cercles et disques peuvent avoir des conditions sur les valeurs qu'ils représentent. Par exemple dans l'image (a) de la figure 6.2, la zone source filtre tous les éléments `<book>` qui ont un attribut *year* dont la valeur est plus grande que 1991, et tels qu'ils aient un fils `<publisher>` avec pour valeur `''Addison-Wesley"`.

Dans la zone de construction, la liaison d'un élément ayant ses sous-éléments liés par des arcs correspond à la projection de ces sous-éléments (dans la requête Q1, seuls le titre et l'année seront retenus). La liaison entre l'élément `<book>` de la zone source à la zone construction indique qu'il y aura dans le résultat tous les éléments `<book>` qui auront été filtrés dans la zone source. Le trapèze contenant la valeur `bib`, signifie que tous ces livres seront contenus dans un unique et nouvel élément `<bib>`. Tous les nouveaux éléments sont représentés par un trapèze.

La deuxième requête de la figure 6.2 dans l'image (b) représente la requête Q3 des « *XML Query Use Cases* » : pour chaque livre de la bibliographie, donner la liste des auteurs et du titre, groupés dans un élément `<result>`. Requête qui s'écrit en XQUERY :

```
<results>
 { for $b in document("www.bn.com/bib.xml")/bib/book
   return <result> { $b/title } { $b/author } </result> }
</results>
```

Dans cet exemple, il n'y a pas de condition sur les valeurs, mais seulement des projections sur les éléments des livres et le renommage de la balise `<book>` en `<result>`. Comme ces deux éléments sont liés alors cela implique la reconstruction d'un nouvel élément `<result>` qui contient les sous-arbres fils de chaque livre. Seuls les auteurs et les titres sont gardés pour chaque élément `<result>`, les autres n'ayant pas été spécifiés sont jetés. Les cercles sous l'élément `<title>` signifient que le texte du titre est gardé. L'astérisque sur l'arc qui part de l'élément `<author>` et rejoint le rectangle vide signifie que tous ses fils à n'importe quelle profondeur seront gardés.

XQBE peut aussi faire des projections à différentes profondeurs. Par exemple considérons la requête qui pour chaque livre ne garde que le titre et les noms de famille des auteurs, et qui s'exprime en XQUERY de cette façon :

```
for $b in //book
return <book> { $b/title } { $b/author/last } </book>
```

où les éléments <author> sont retirés du résultat, pour ne garder que le sous-éléments <last>. L'élément <author> est représenté par des lignes en pointillés. XQBE permet d'exprimer de façon intuitive en annotant des arbres XML une requête XQUERY.

Nous préférons garder pour notre outil graphique l'esprit tableau de QBE, pour capturer des éléments, plutôt que l'annotation d'arbres comme XQBE. Ces éléments pouvant être réutilisés dans d'autres tableaux, et favoriser la compositionnalité de requêtes.

6.3 Présentation de PBE

Nous présentons PBE (pour « *Pattern by Example* ») qui est l'interface graphique développée dans le cadre de cette thèse, et qui est spécialement adaptée à la manipulation de documents XML en présence d'une DTD. L'idée principale est d'aider l'utilisateur lors de l'écriture d'une requête, en se servant des types définis dans les DTD, et du système de type de ℂDuce. À chaque moment, l'utilisateur sera guidé par le type des valeurs qu'il construit. Il y a aussi la possibilité de construire des requêtes plus complexes en composant des requêtes plus simples.

Cette interface permet :

- de guider un utilisateur lors de l'écriture de requêtes
- de générer des requêtes optimisées dans le langage ℂQL
- de les évaluer
- d'être indépendant de ℂDuce

6.3.1 Visite guidée

Pour construire une requête, nous avons besoin d'une DTD représentée[1] par des types ℂDuce qui nous guideront pour la construction de tableaux. Nous définissons deux types de tableaux, les *tableaux de filtrage* pour capturer des éléments dans

[1] Grâce au programme `dtd2cduce` (disponible sur `www.cduce.org`) qui permet à partir d'une DTD d'obtenir des types ℂDuce correspondants.

des variables et les *tableaux de construction* pour reconstruire une séquence de valeurs XML à partir de variables définies dans les tableaux de filtrage.

Nous définissons ici la DTD utilisée pour la visite guidée, et nous donnons un document XML.

6.3.1.1 DTD et valeur XML de référence

Nous ne considérons que des DTD sans alternative dans les expressions régulières. Nous le verrons par la suite, les tableaux de filtrage ne tiennent pas compte de l'union d'entités. Il s'agit d'un problème graphique de représentation de cette union, qui sera étudié dans un futur travail.
C'est pourquoi nous allons changer la DTD de référence, en supprimant l'entité `Editor`. Cette nouvelle DTD, et le document qui serviront de base pour les exemples sont donnés dans les figures 6.3 et 6.4.

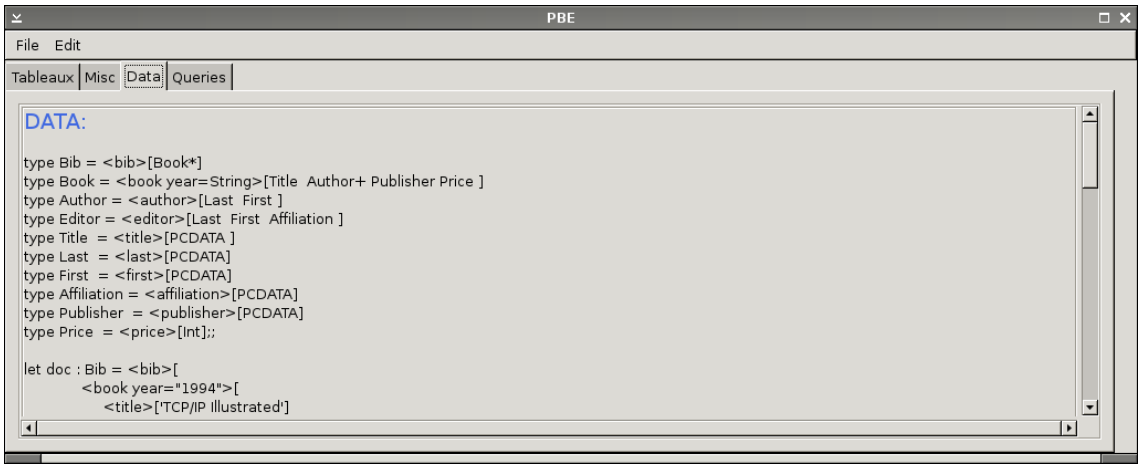


FIG. 6.3 – DTD de référence pour PBE

Les tableaux de PBE permettent d'exprimer de nombreuses opérations de requêtes, nous allons en donner quelques exemples.

Sélection

6.3.1.2 Q1 : Retourner tous les livres

Supposons que le document source soit représenté par la variable `doc`, l'utilisateur pour interroger ce document devra choisir cette variable dans la partie gauche d'un tableau de filtrage (comme représenté sur la figure 6.5) parmi les différentes racines de persistance (ou variables définies dans un autre tableau de filtrage).

Dans cet exemple, PBE connaît le type de `doc` (qui est une racine de persistance) qui est *Bib*, et sait donc que ce type contient une séquence d'éléments de type

```
<bib>
   <book>
       <title>TCP/IP Illustrated</title>
       <author>
          <last>Stevens</last>
          <first>W.</first>
       <publisher>Addison-Wesley</publisher>
       <price>65</price>
   </book>
   <book>
        <title>Advanced Programming in the Unix environment</title>
        <author>
          <last>Stevens</last>
          <first>W.</first>
        </author>
        <publisher>Addison-Wesley</publisher>
        <price>65</price>
   </book>
   <book>
        <title>Data on the Web</title>
        <author>
          <last>Abiteboul</last>
          <first>Serge</first>
        </author>
        <author>
          <last>Buneman</last>
          <first>Peter</first>
        </author>
        <author>
          <last>Suciu</last>
          <first>Dan</first>
        </author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39</price>
   </book>
</bib>
```

FIG. 6.4 – Document XML de référence pour PBE

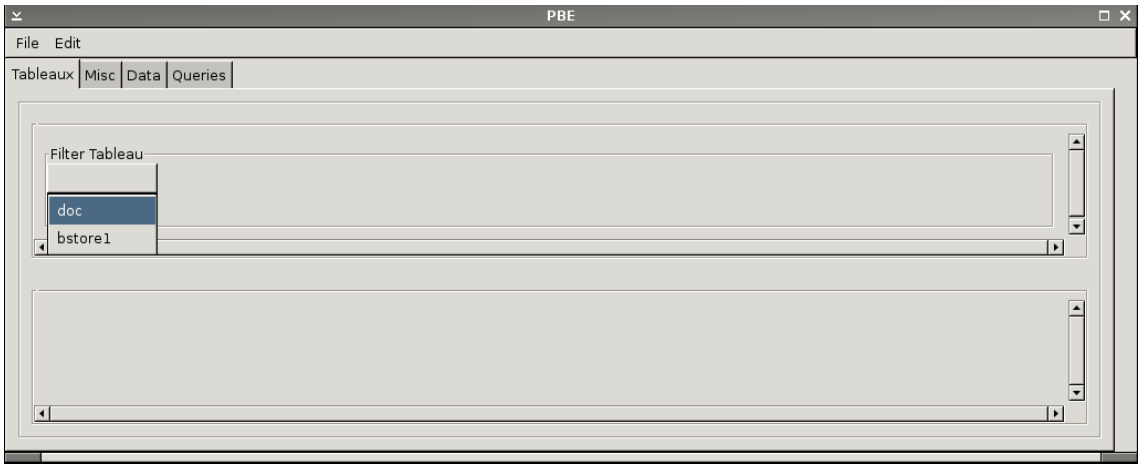


FIG. 6.5 – Création d'un tableau de filtrage

$Book$. Dans la partie gauche de la première ligne, sera inscrit le type de la variable doc, à savoir Bib, puis une case avec le symbole # (pour capturer le nom de la balise[2]) et ensuite, une série de types, à la XDuce, correspondant aux types contenus par Bib, ici $Book*$. À partir de la seconde ligne, chaque ligne servira à déclarer des variables pour les éléments à capturer, à partir d'une même racine[3] qui sera inscrite dans la première colonne (ici doc comme dans la figure 6.6).

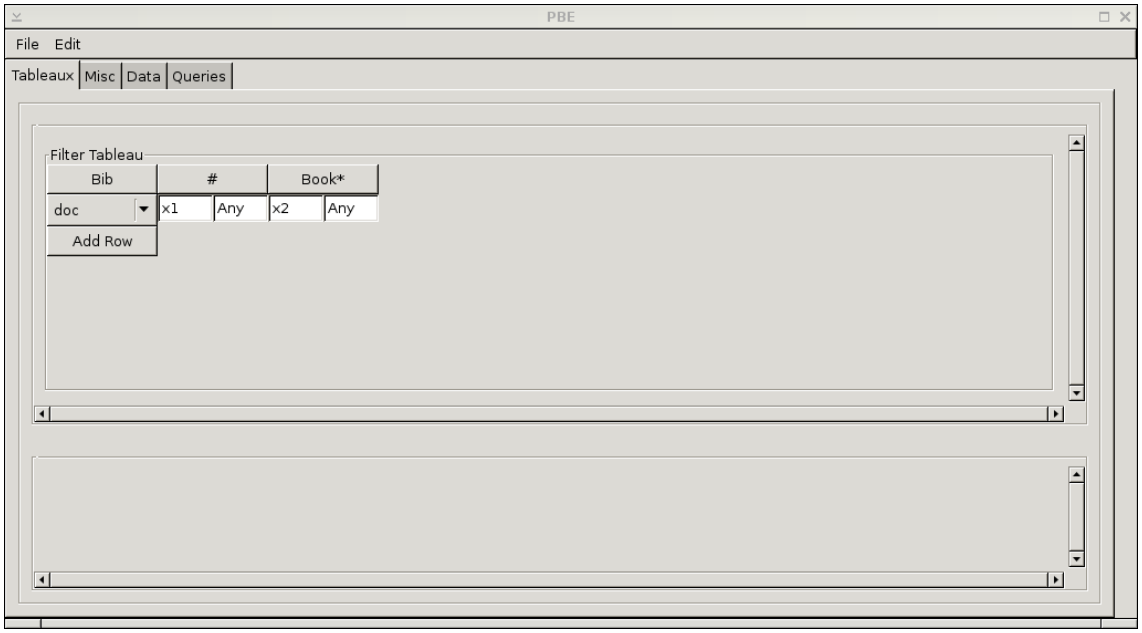


FIG. 6.6 – Tableau de filtrage créé portant sur la variable doc

Dans l'exemple qui suit (figure 6.7), l'utilisateur définit une nouvelle variable books en la posant dans la case en dessous du type $Book*$, et signifie que dans cette variable sera capturée la séquence des éléments de type $Book$ de doc. Nous ne modifions pas la seconde case, et laissons la valeur Any. Cette case permet d'exprimer des contraintes de types, sur ces les éléments de books, nous les verrons dans la suite.

Nous avons besoin de définir une reconstruction des éléments capturés, comme dans XQBE avec la partie « *construct* », reconstruction que nous appellerons tableau de construction. Un tel tableau est composé de deux lignes. Dans la seconde ligne, nous marquerons en premier la variable donnée pour cette requête afin de l'utiliser éventuellement dans d'autres tableaux de construction : puis toutes les variables pour construire un nouvel élément XML qui aura pour balise, la valeur ou le contenu de la variable dans la première case de la première ligne. Les cases restantes de cette première ligne sont les types respectifs de chaque variable de

[2]C'est utile, par exemple, dans le cas où on utilise les types ℂDuce, et qu'on dispose d'un type pouvant contenir différentes balises. ex : `type AB = <('a|'b)>[Any*]`.

[3]Nous aurons ensuite le choix entre ces nouvelles variables et les racines

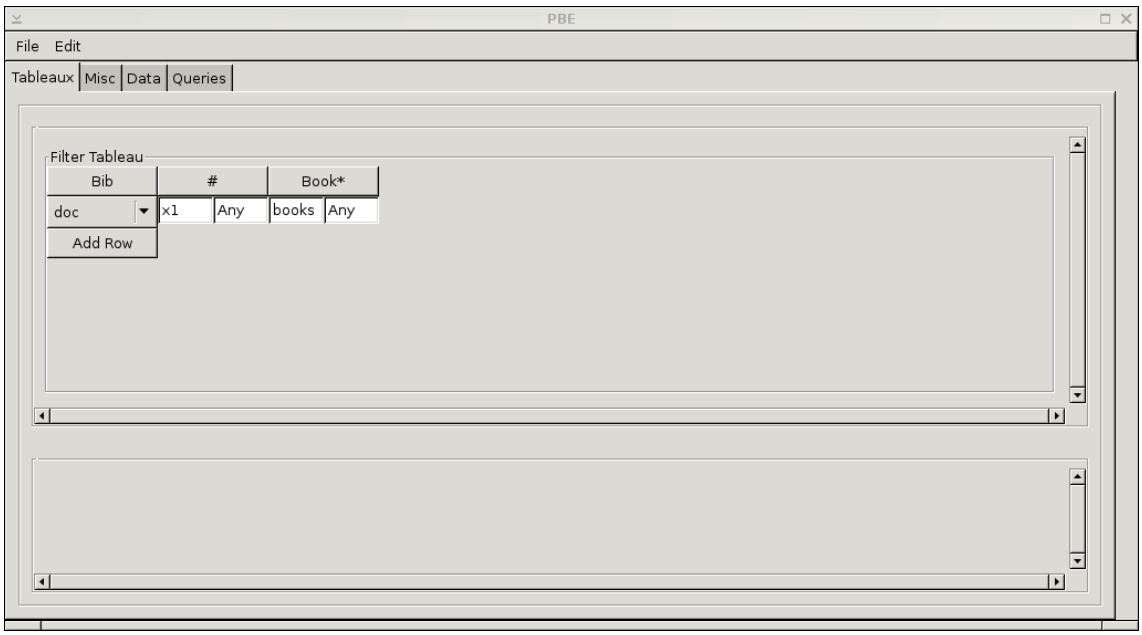


FIG. 6.7 – Ajout de la variable `books` dans le tableau de filtrage.

la seconde ligne. PBE permet d'attribuer une variable à un tableau de construction, dans le but de les composer, comme nous le verrons dans la suite. Dans cet exemple (figure 6.8), nous utiliserons la variable q1 pour représenter le résultat de ce tableau de construction. Ce dernier est une requête, telle que la balise de chaque élément ait une balise `<result>` et qu'elle contienne l'ensemble des livres représenté par la variable `books`.

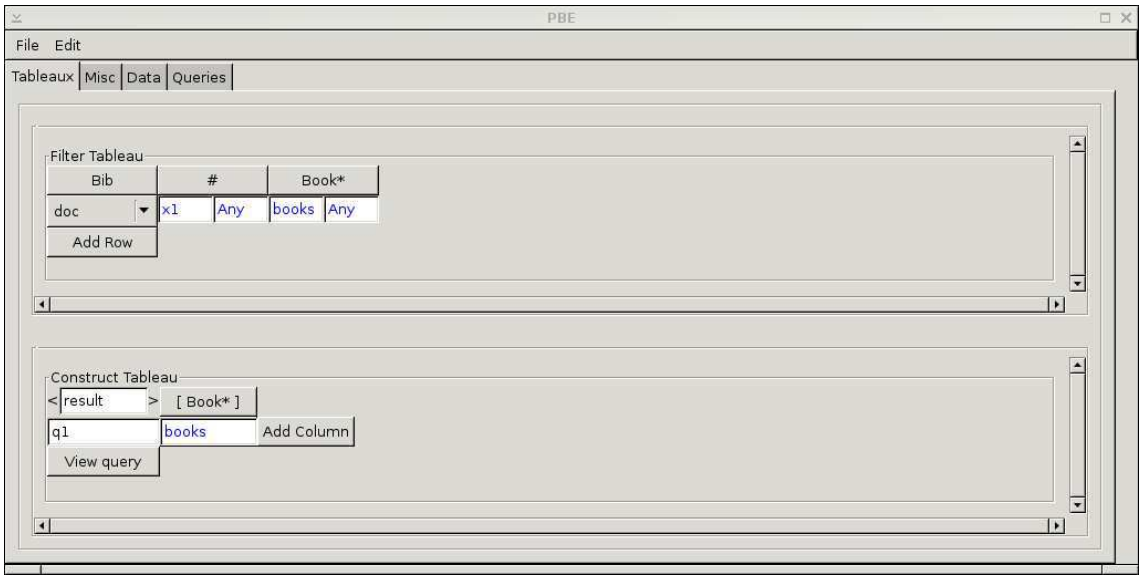


FIG. 6.8 – Création du tableau de construction q1

En cliquant sur le bouton « View query », PBE va calculer, à partir de ces deux tableaux corrects, la requête ℂQL correspondante donnée dans la figure 6.9.

PBE inférera aussi que le résultat de la requête aura pour type

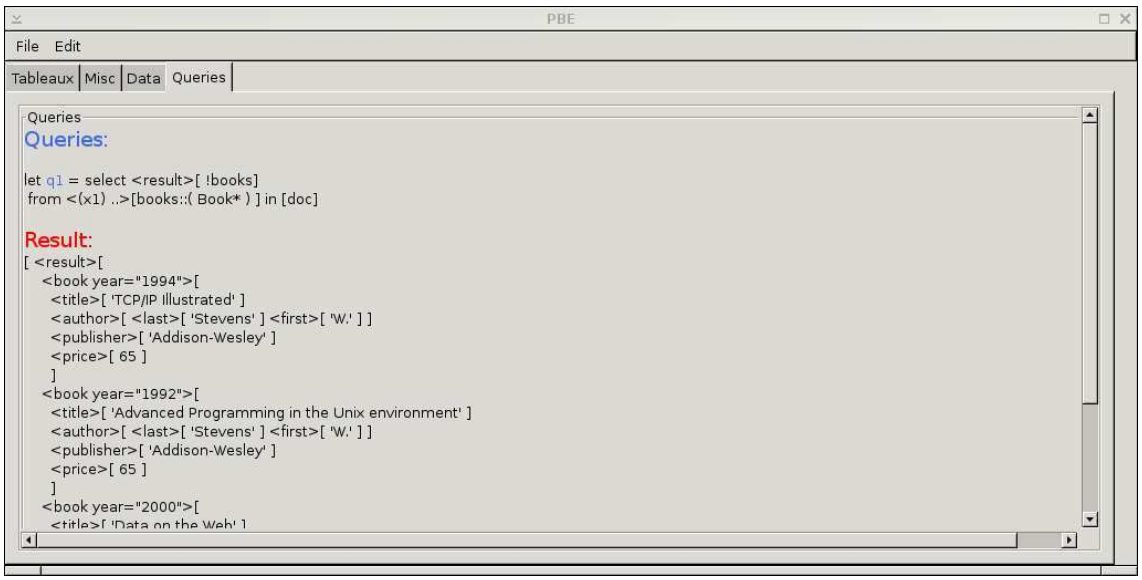


FIG. 6.9 – La requête q1 déduite en ℂQL.

`[<result>[Book*] *]` ; et pourra se servir ultérieurement de ce type quand `q1` sera utilisée dans d'autres tableaux de construction.

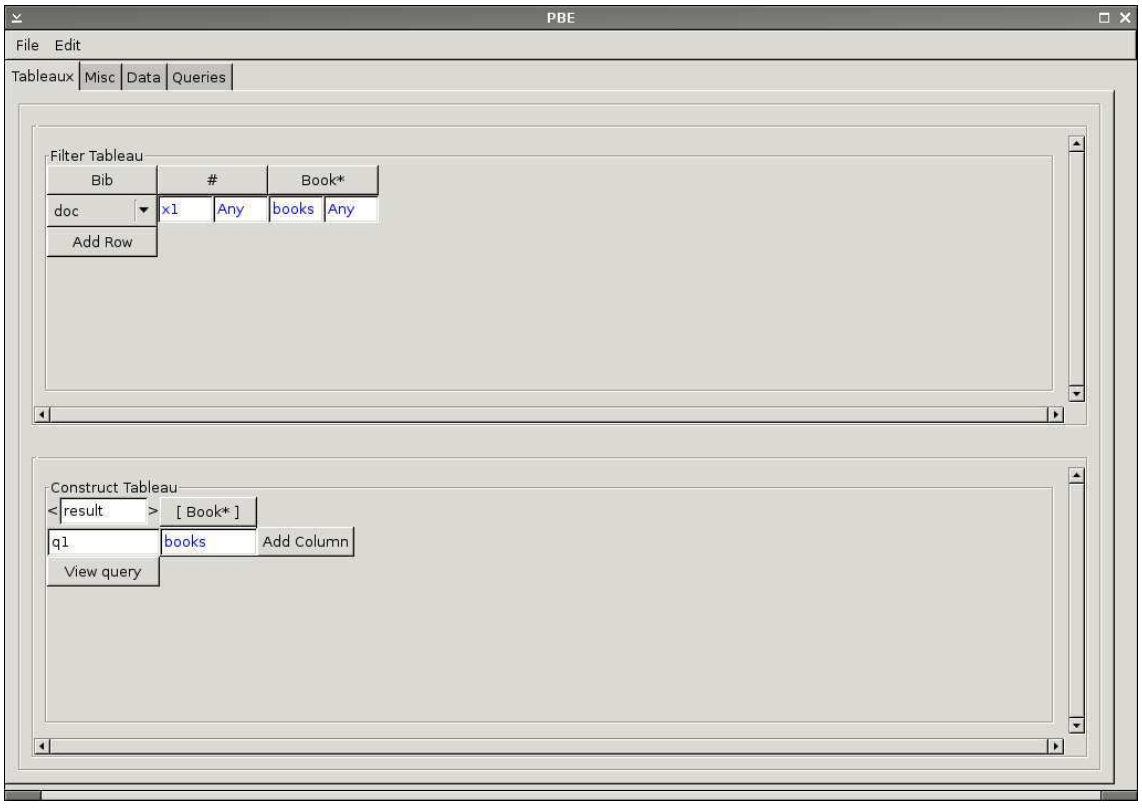


FIG. 6.10 – Retourner tous les livres

6.3.1.3 Q2 : renvoyer tous les titres

Nous pouvons aussi capturer des éléments de chaque livre en itérant sur la variable `books`. Par exemple, pour capturer tous les titres, nous devons ouvrir un

nouveau tableau de filtrage qui porte sur la variables `books`, et déclarer une variable `title`. Dans la reconstruction de la requête q_2 nous utiliserons cette variable (dans la figure 6.11).

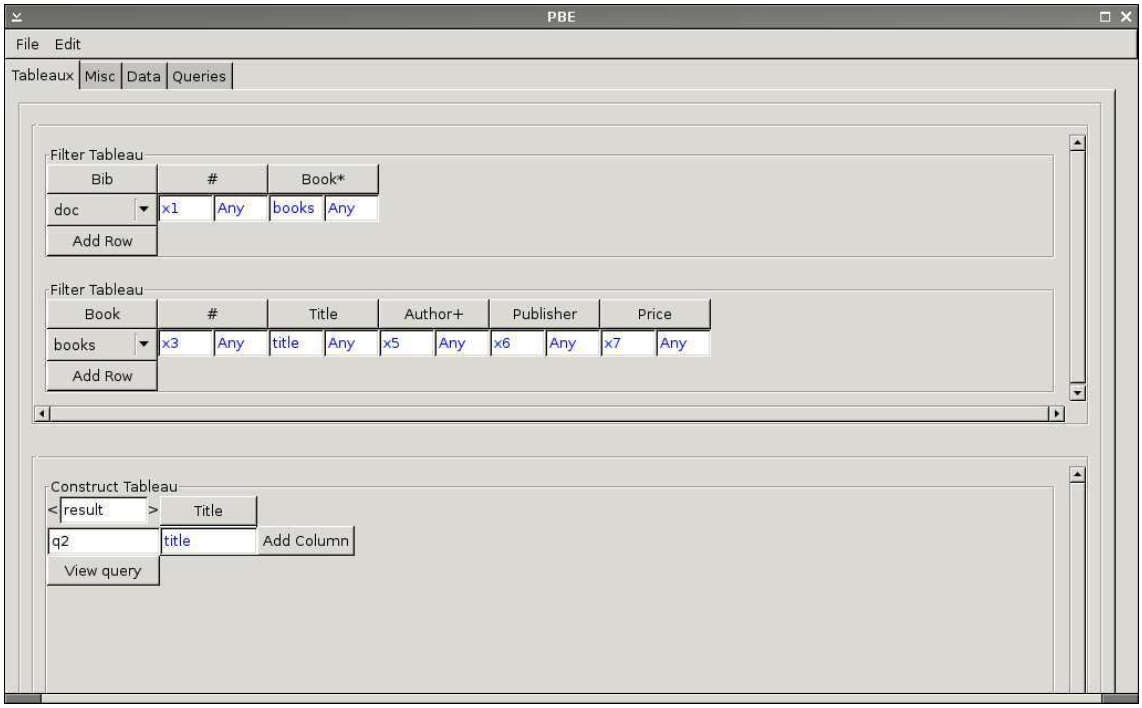


FIG. 6.11 – Retourner tous les titres de livres

En cliquant sur le bouton *"View Query"*, le code de la requête est montré avec le résultat sur le document (dans la figure 6.12).

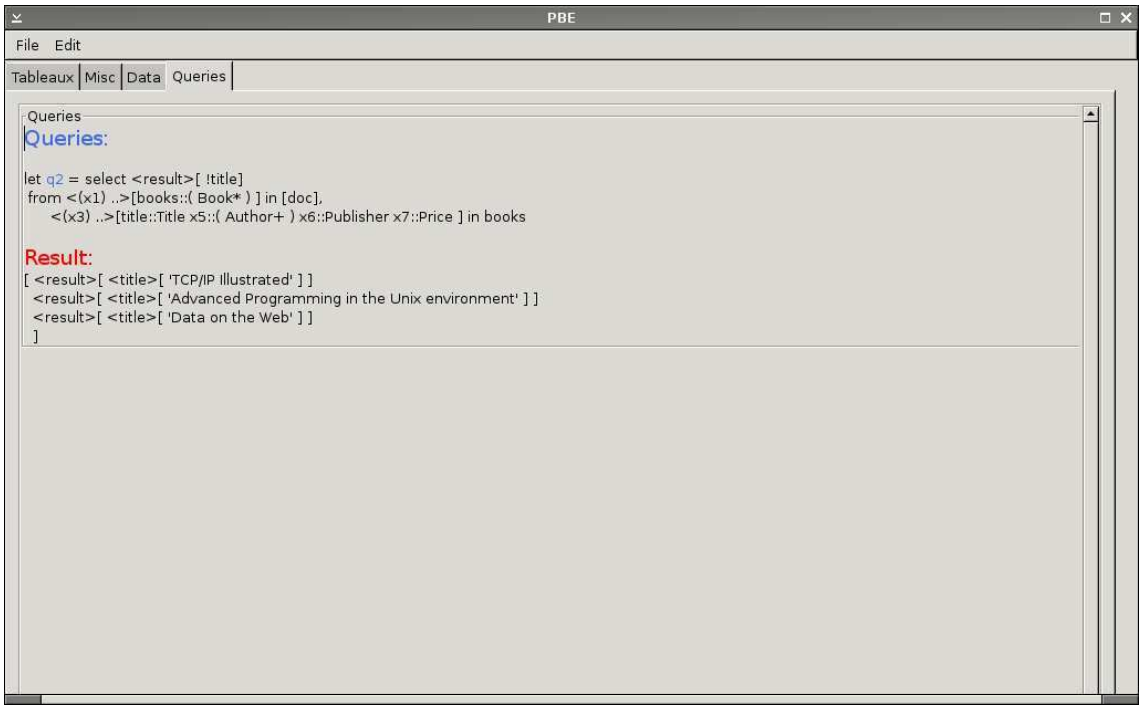


FIG. 6.12 – Requête CQL et la réponse.

6.3.1.4 Q4 : Renvoyer tous les titres avec les noms de famille des auteurs dans une nouvelle balise.

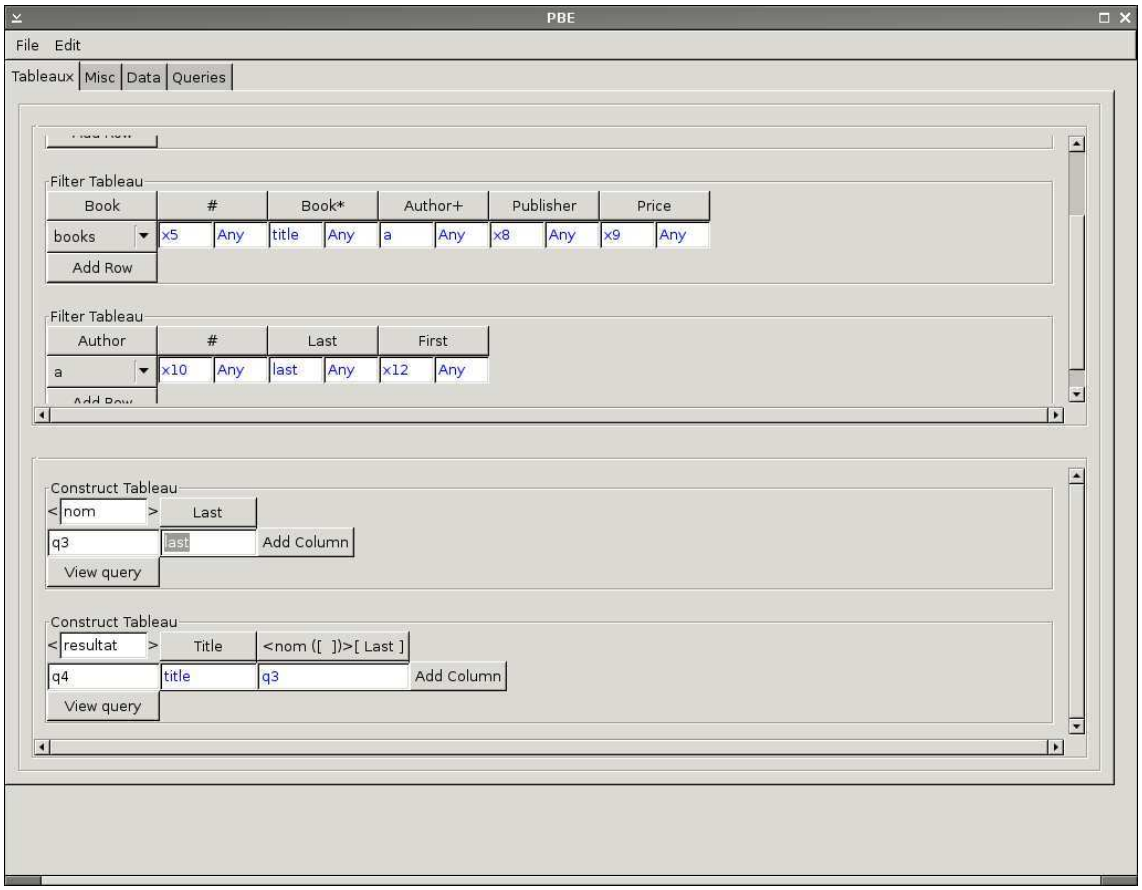


FIG. 6.13 – Exemple de requête imbriquée

Dans la figure 6.13 nous avons un exemple de construction de requêtes imbriquées. Dans le tableau qui porte sur `books` nous définissons deux variables `title` et `a`, pour associer les auteurs au titre. Puis nous capturons pour chaque auteur de `a`, la variable `last`. Pour pouvoir encapsuler chaque élément `<last>` dans une nouvelle balise `<nom>`, nous définissons un tableau de construction appelé `q3`. Ce tableau peut être réutilisé dans un autre tableau de manière à pouvoir réutiliser son résultat. Ici, dans le tableau de construction de la requête `q4`, nous demandons le titre d'un livre, et le résultat de la requête `q3` pour ce titre.

En exécutant la requête q3 on obtient le code et le résultat suivants :

```
let q3 = select <nom>[ !last]
 from <(x1) ..>[books::( Book* ) ] in [doc],
      <(x5) ..>[title::Title a::( Author+ ) x8::Publisher x9::Price ] in books,
      <(x10) ..>[last::Last x12::First ] in a
Result:
[ <nom>[ <last>[ 'Stevens' ] ]
```

```
<nom>[ <last>[ 'Stevens' ] ]
<nom>[ <last>[ 'Abiteboul' ] ]
<nom>[ <last>[ 'Buneman' ] ]
<nom>[ <last>[ 'Suciu' ] ]
]
```

Cette requête renvoie la séquence de tous les noms de famille de chaque auteur avec une nouvelle balise <nom>. Pour reconstruire la requête q4 comme elle fait appel à la requête q3, nous ne pouvons pas juste copier le code de la requête q3 à l'endroit où la variable q3 est utilisée, car alors on perdrait la relation qui lie le titre et l'auteur. PBE reconstruit alors différemment la requête suivant un algorithme qui sera expliqué dans les sections suivantes pour permettre de ne pas perdre la sémantique donnée aux requêtes, et également de reconstruire une requête qui aie le moins de motifs possibles. Car on a vu dans l'étude des performances en suivant l'approche micro-benchmarks, que la répétition de motifs identiques pouvait ralentir l'exécution d'une requête. L'algorithme de traduction d'une requête PBE en une requête CQL permettra de trouver une requête utilisant le moins de motifs possibles.

On a ainsi la requête q4 en $\mathbb{C}$QL donnée dans la figure 6.14, où la variable q3 a disparu, et les motifs modifiés par rapport à ceux qui capturaient seulement le titre de la figure 6.12

6.3.1.5 Q5 : Tableaux à plusieurs lignes

Il existe la possibilité d'exprimer des contraintes différentes en utilisant une nouvelle ligne d'un même tableau. Toutes les variables doivent être identiques dans ces lignes. Dans l'exemple de la figure 6.15, on demande à ce que le titre d'un livre commence par la lettre 'T' exprimé par l'expression régulière `['T' _*]`, ou par la lettre 'D'. Ce qui est intéressant dans cette requête est que lors de la reconstruction, lorsque l'utilisateur donne la variable `Texte`, PBE donne le type de cette variable inféré par la connaissance qu'il en a, en fonction de la DTD, et des contraintes de types posées dans les cases correspondantes. Ici on voit que la variable `Texte` a pour type : `['D'|'T' Char*]`.

La requête déduite des tableaux de la figure 6.15 est donnée dans la figure 6.16

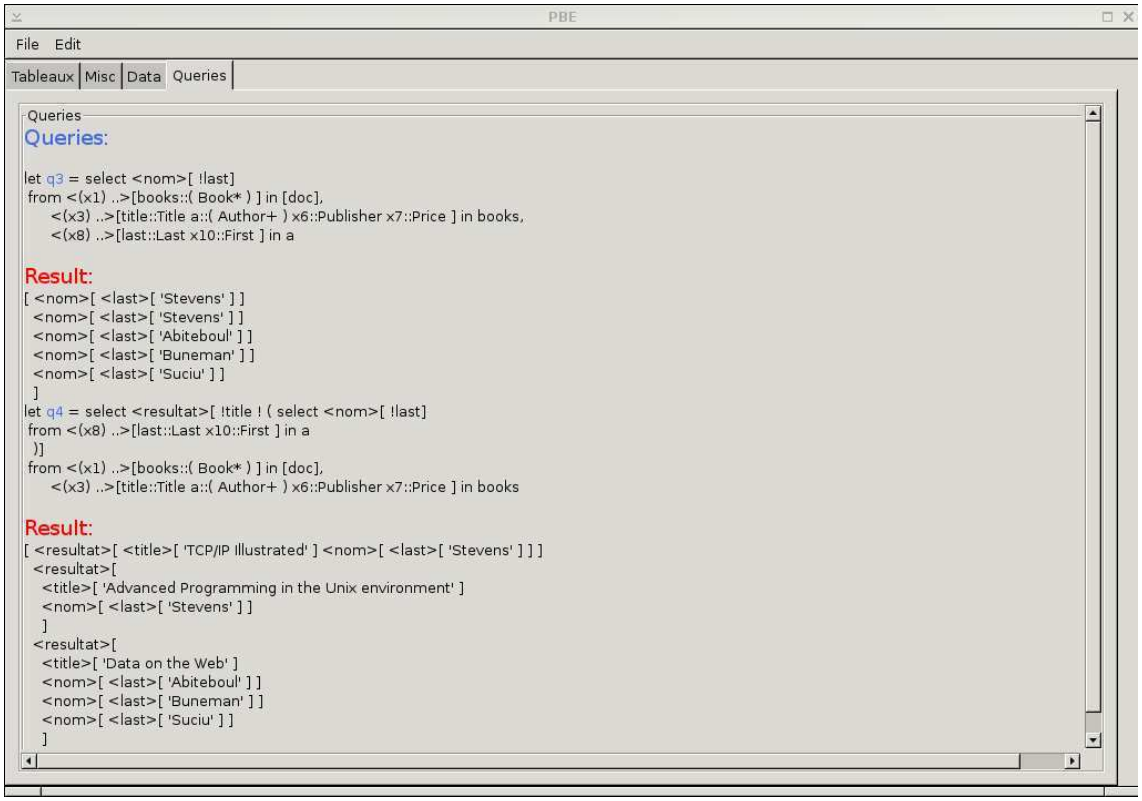


FIG. 6.14 – Les requêtes q3 et q4 déduites en ℂQL.

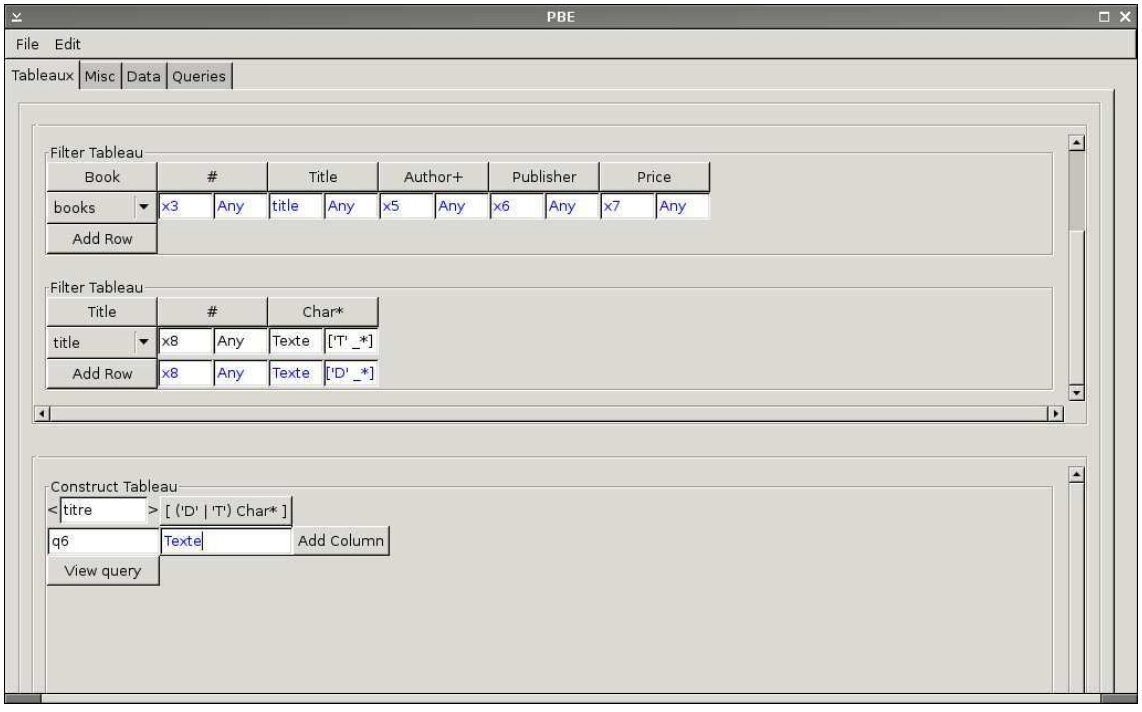


FIG. 6.15 – Exemple de tableaux à plusieurs lignes

6.4 Description formelle de PBE

Nous présentons dans la suite la description formelle de PBE. Dans un premier temps nous avons besoin de construire les tableaux de filtrage, et notamment les

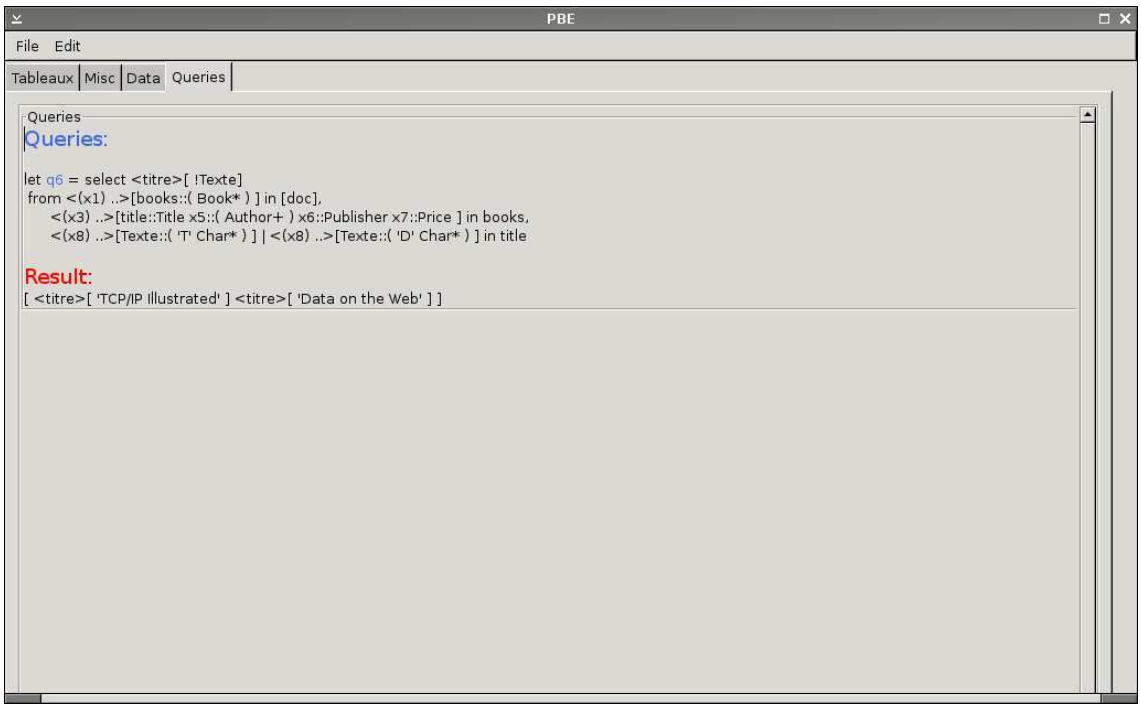


FIG. 6.16 – Résultat de la requête

expressions de types qui seront annotées en en-tête d'un tableau. Pour cela nous avons besoin de définir *le produit maximal* d'une expression régulière de types qui permet de découper une expression régulière représentant la séquence des fils des éléments de type XML. Nous donnons alors une définition des types ℂDuce considérés dans ce cadre, comprenant le type XML construits à partir d'expressions régulières de séquences. Ensuite nous présentons la syntaxe des tableaux de filtrage et de construction, et des boîtes de condition. Puis nous étudierons un algorithme de traduction de requêtes PBE en requêtes ℂQL permettant de donner une sémantique à ces requêtes.

6.4.1 Définitions

TypeXML	: :=	<tag {Attlist}>[TRegexp] | Identifieur
Attlist	: :=	Att Attlist | ϵ
Att	: :=	Identifieur= BasicType
TRegexp	: :=	TypeXML | TRegexp TRegexp
	|	TRegexp* | TRegexp+ | TRegexp ? | ϵ

FIG. 6.17 – Définition des types ℂDuce utilisés pour les en-tête de tableaux de filtrage

Ces types seront utilisés par nos tableaux. Les TRegexp nous serviront dans la

partie droite en en-tête des tableaux de filtrage, c'est pour cela, que nous avons les caractères `*`, `+?`, pour pouvoir représenter les expressions régulières de types ℂDuce. Notons l'absence de l'union, et la similarité des TRegexp avec les types de XDuce.

Définition 6.4.1 *Produit d'expressions régulières de séquences :*

Un TRegexp R est un produit d'expressions régulières de séquences si il existe R_1, R_2 tels que :

$R \equiv R_1\ R_2$ *et* $R_1 \not\equiv R_2$ *(syntaxiquement)*

Définition 6.4.2 *Produit maximal d'expressions régulières de séquences :*

Soit R un TRegexp, nous appellerons produit maximal de R toutes les expressions de la forme :

$R \equiv R_1\ R_2\ \ldots\ R_k$ *où* R_i *n'est pas un produit d'expressions régulières de séquences.*

Par exemple considérons les types XML définis ainsi :

```
type A = <a> [B* B+ C B ]
type B = <b b=int> [A C A ]
```

L'expression régulière de séquence du premier type `A` n'est pas un produit maximal d'expressions régulières de séquences, car nous avons la séquence `B* B+` qui n'est pas un produit d'expressions régulières de séquences. Tandis que le type B l'est. Mais le type A peut se réécrire en :

```
type A = <a> [B+ C B ]
```

qui est alors un produit maximal.

Syntaxe des expressions

Il existe différentes expressions :

- *Les variables* qui peuvent représenter les noms des racines de persistance, ou les séquences capturées dans des tableaux de filtrage, ou les noms de requêtes définies dans les tableaux de construction.
- *Les valeurs* qui sont des valeurs XML.
- *Les expressions régulières* qui sont utilisées dans les tableaux de filtrage pour poser des conditions, et qui sont celles de ℂDuce.

6.4.2 Syntaxe des tableaux

Il existe deux types de tableaux :

- le tableau de filtrage, pour capturer dans des variables des sous-séquences d'éléments définies par des variables, et pour poser des conditions au moyen d'expressions régulières sur ces séquences. Sur l'exemple suivant, si nous considérons la variable `books` de type `book*`, alors nous avons les TRegexp `title,author+,publisher` et `price` comme en-tête, et nous pouvons déclarer sous chacune les variables x_i et les contraintes de type t_i qui vont capturer tous les éléments qui respectent la contrainte de type.

book	#	title	author+	publisher	price
books	(x_1, t_1)	(x_2, t_2)	(x_3, t_3)	(x_4, t_4)	(x_5, t_5)

- le tableau de construction, pour reconstruire une requête à partir de variables déjà définies dans des tableaux de filtrage ou d'autres tableaux de construction. Sur l'exemple suivant, nous pouvons construire une séquence qui contient une balise `<result>` et qui se sert des variables précédemment définies et que nous appelons q, de manière à pouvoir être réutilisée et ainsi réimbriquer plusieurs tableaux de construction.

<result>	title	price
q	x_2	x_5

Nous donnons précisément leur syntaxe dans la partie suivante.

6.4.2.1 Le tableau de filtrage

Définition 6.4.3 *Soit t un type XML, le tableau de filtrage associé à t est :*

t	#	a_1	$\cdots$	a_k	R_1	$\cdots$	R_n
y	$(x_0, t_{1,0})$	$(x_1, t_{1,1})$	$\cdots$	$(x_k, t_{1,k})$	$(x_{k+1}, t_{1,k+1})$	$\cdots$	$(x_{k+n}, t_{1,k+n})$
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$
y	$(x_0, t_{m,0})$	$(x_1, t_{m,1})$	$\cdots$	$(x_k, t_{m,k})$	(x_{k+1}, t_{k+1})	$\cdots$	$(x_{k+n}, t_{m,k+n})$

où

- y *est une variable liée de type* $[t*]$ *ou une racine de persistance de type* t,
- $t =$ *<tag* $\{a_1 = t_1 \ldots a_k = t_k\}$*>[R]*,
- R *est un produit maximal, tel que* $R = R_1 \ \ldots \ R_n$,
- x_j *sont des variables libres* $(j = 0 \ldots k+n)$
- $t_{i,j}$ *sont des expressions régulières* $\mathbb{C}$Duce $(i = 1 \ldots m, j = 0 \ldots k+n)$.

Considérons la première ligne . Pour chaque élément de la séquence dénotée par y, de nouvelles variables $\{x_0, ..., x_{k+n}\}$, vont être définies telles que :

- x_0 représente la balise de l'élément, et $t_{1,0}$ soit la contrainte de type associées à x_0 qui permet de filtrer une partie des balises capturées.
- $x_1 \dots x_k$ représentent respectivement les valeurs des attributs $\{a_1 \dots a_k\}$, tels que les $t_{1,1} \dots t_{1,k}$ soient les contraintes de types $\mathbb{C}$Duce associées aux variables $x_1 \dots x_k$, et qui permettent de filtrer une partie des attributs capturés.
- $x_{k+1} \dots x_{k+n}$ représentent respectivement les valeurs de $\{R_1 \dots R_n\}$, tels que les $t_{1,k+1} \dots t_{1,k+n}$ soient les contraintes de types $\mathbb{C}$Duce associées aux variables $x_{k+1} \dots x_{k+n}$, et qui permettent de filtrer une partie des valeurs des fils de l'élément considéré.

Dans la suite, chaque ligne d'un tableau de filtrage sera représenté par l'objet : $FT(y|tag|k|(x_0, t_{i,0})|(x_1, t_{i,1}) \dots (x_k, t_{i,k})|(x_{k+1}, t_{i,k+1}) \dots (x_{k+n}, t_{i,k+n}))$, où k représente le nombre d'attributs, et i la ligne du tableau.

6.4.2.2 Le tableau de construction

Définition 6.4.4 *Soient des variables $x_{i,j}$ et une variable ou une valeur tag, un tableau de construction est représenté par :*

tag	a_1	$\cdots$	a_k	R_1	$\cdots$	R_n
y	x_1	$\cdots$	x_k	x_{k+1}	$\cdots$	x_{k+n}

où

- *les R_i sont respectivement les types pour chaque x_{k+i} ($i = 1 \dots n-k$),*
- *tag est une variable représentant une balise, ou une chaîne de caractères.*
- *et y une variable libre de type $[t_0*]$, tel que le type t_0 est calculé par PBE comme étant $t_0 =$ <tag $\{a_1 = t_1 \dots a_k = t_k$>} [$R_1$ ... R_n]*

Le tableau de construction est utilisé pour construire des séquences d'éléments XML avec une nouvelle balise, à partir de variables capturées précédemment dans le tableaux de filtrage (ou de variables dénotant des racines de persistance), ou des variables dénotant des requêtes précédemment construites.

Dans la suite le tableau de construction sera représenté par l'objet $CT(y|tag|k|(a_1, x_1) \dots (a_k, x_k)|x_{k+1} \dots x_{k+n})$, où k représente le nombre d'attributs.

6.4.2.3 Boîte de condition

La boîte de condition est directement inspirée de QBE, et est utilisée pour poser des contraintes qui peuvent ou non être exprimées par des expressions régulières dans les tableaux de filtrage. Celle-ci n'a pas été montrée dans les exemples, néanmoins elle est de la forme suivante :

CONDITION BOX
e_1
$\vdots$
e_n

où e_i est une comparaison $\mathbb{C}$Duce entre deux variables, ou entre une variable et une valeur $\mathbb{C}$Duce. Cette boîte est très utile pour exprimer les conditions de jointures. Par exemple, si les variables t_1 et t_2 sont définies dans des tableaux de filtrage, et que nous voulons leur égalité pour faire une jointure, alors nous devons placer la condition $t_1 = t_2$ dans une ligne de la boîte de condition (comme dans le cas de la figure 6.29 à la page 157).

Dans la suite une ligne d'une condition de la boîte de condition sera représentée par l'objet $CB(e)$.

6.4.3 Sémantique

Définition 6.4.5 *Une requête PBE est composée d'une ou plusieurs racines de persistance, d'un ou plusieurs tableaux de filtrage, et d'un ou plusieurs tableaux de construction, et d'éventuellement une boîte de condition.*

Pour construire une requête, nous avons besoin de capturer des sous-éléments à partir de racine de persistance, et de les mettre dans des variables. Ce qui est possible par des tableaux de filtrage. Les requêtes seront construites par des tableaux de construction.

Mais ce n'est pas une condition suffisante, il faut vérifier que chaque variable utilisée pour la reconstruction de la requête est définie précédemment.

Nous définissons les fonctions DV et FV, qui calculent respectivement les variables liées, et les variables libres d'un tableau. Ces fonctions nous serviront lors de la présentation de l'algorithme de reconstruction de la requête en $\mathbb{C}$QL.

Définition 6.4.6 *Soit O un objet qui est un $CT(_)$, un $FT(_)$ ou un $CB(_)$, nous définissons $DV(O)$ par :*

$DV(CT(y|tag|k|(a_1, x_1)\dots(a_k, x_k)|x_{k+1}\dots x_{k+n})) = \{y\}$

$DV(FT(y|tag|k|(x_0, t_{i,0})|(x_1, t_{i,1})\dots(x_k, t_{i,k})|(x_{k+1}, t_{i,k+1})\dots(x_{k+n}, t_{i,k+n}))) = \{x_0\dots x_{k+n}\}$

$DV(CB(e)) = \varnothing$

Définition 6.4.7 *Soit O un objet qui est un $CT(_)$, un $FT(_)$ ou un $CB(_)$, nous définissons $FV(O)$ par :*

$FV(CT(y|tag|k|(a_1, x_1)\dots(a_k, x_k)|x_{k+1}\dots x_{k+n})) = \{x_1\dots x_{k+n}\}$

$FV(FT(y|tag|k|(x_0,t_{i,0})|(x_1,t_{i,1})\dots(x_k,t_{i,k})|(x_{k+1},t_{i,k+1})\dots(x_{k+n},t_{i,k+n}))) = \{y\}$
$FV(CB(e)) = var(e)$

où la fonction $var()$ est celle de la définition 3.2.1 à la page 55.

6.4.4 Algorithme de traduction d'une requête PBE en une requête ℂQL

La sémantique de PBE est donnée par un algorithme de traduction d'une requête PBE en une requête ℂQL. Pour un souci de compréhension de l'algorithme nous présenterons les cheminements qui conduisent à cet algorithme. Nous allons d'abord étudier un algorithme de reconstruction de requêtes non-imbriquées sans condition, puis nous verrons qu'il génère une forme de redondance que nous supprimerons ; nous verrons ensuite le cas pour les requêtes imbriquées, c'est-à-dire pouvant être composées de plusieurs tableaux de construction, puis nous rajouterons les conditions.

6.4.4.1 Cas de requêtes non-imbriquées et sans condition

Dans une première partie nous considérerons les cas de requêtes non-imbriquées

Nous définissons trois environnements. Le premier que nous appellerons $\mathcal{P}$ contient l'ensemble des racines de persistance. Le suivant $\mathcal{F}$ qui contient l'ensemble des lignes de tableaux de filtrage, les FT, et le dernier $\mathcal{C}$ qui contient l'ensemble des CT, les tableaux de construction.

Les règles d'inférence pour ce cas de traduction sont présentées dans la figure 6.18. La règle d'inférence *R2* associée avec le jugement $\vdash_s$ va construire la clause `select` d'une requête ℂQL. Comme nous avons pour l'instant considéré le cas où nous n'avons pas de requêtes imbriquées, seules les variables sont présentes dans un CT et devraient être définies dans $\mathcal{F}$. Le jugement :

$\mathcal{F},\mathcal{C} \vdash_s x \rightarrow e$

produira une expression e ℂDuce à partir de la variable x. Il s'agit du point d'entrée de l'algorithme, ainsi pour une requête appelée q1, on évaluera $\mathcal{F},\mathcal{C} \vdash_s x1 \rightarrow e$, et l'expression e sera la requête ℂQL correspondant à la traduction.

La règle d'inférence *R6* est pour le cas où il n'existe pas de variable définie dans un CT de $\mathcal{C}$. Comme l'algorithme ne peut pas évaluer une requête qui n'est pas définie, alors on renvoie la valeur Ω.

Dans la règle *R2*, nous construisons la clause `from` en appelant le jugement $\vdash_f$ qui est défini ainsi :

$$\frac{CT(x|tag|k|(a_1,x_1)\ldots(a_k,x_k)|x_{k+1}\ldots x_{k+n}) \in \mathcal{C} \quad \mathcal{F} \vdash_f x_i \rightarrow l_i \quad i = 1\ldots k+n}{\mathcal{F},\mathcal{C} \vdash_s x \rightarrow \texttt{select < } tag\ a_1\texttt{=}x_1\ \ldots\ a_k\texttt{=}x_k\texttt{>[!}x_{k+1}\ldots\ \texttt{!}x_{k+n}\texttt{] from } l_1,\ldots,l_{k+n}} \ (R2)$$

$$\frac{\nexists ct \in \mathcal{C}, q \in FV(ct)}{\mathcal{F},\mathcal{C} \vdash_s x \rightarrow \Omega} \ (R6)$$

$$\frac{\nexists ft \in \mathcal{F}, x \in DV(ft)}{\mathcal{F} \vdash_f x \rightarrow \Omega} \ (F2)$$

$$\frac{\begin{array}{l}\exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \\ y \in \mathcal{P} \quad \text{pattern}(ft) \rightsquigarrow p\end{array}}{\mathcal{F} \vdash_f x \rightarrow p\ \texttt{in [}y\texttt{]}} \ (F3)$$

$$\frac{\begin{array}{l}\exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \\ y \notin \mathcal{P} \quad \text{pattern}(ft) \rightsquigarrow p \qquad \mathcal{F}\backslash ft \vdash_f y \rightarrow l\end{array}}{\mathcal{F} \vdash_f x \rightarrow l\ \texttt{,}\ p\ \texttt{in}\ y} \ (F4)$$

FIG. 6.18 – Règles d'inférence pour les requêtes PBE non imbriquées et sans boîte de condition.

$\mathcal{F} \vdash_f x \rightarrow l$

Cela va produire une liste l de « motif `in` expression » qui correspondent à la clause `from`. La fonction pattern(ft) (définie plus loin dans la figure 6.19) va calculer le motif correspondant à un ft et sera de la forme p `in` e. Il y a deux possibilités soit la variable y (portant sur le ft) est une racine de persistance, soit elle ne l'est pas. Dans le premier cas, qui correspond à la règle (*F3*), nous pouvons arrêter la recherche de y, puisqu'elle est définie. Dans la règle (*F4*), y n'est pas une racine de persistance, donc nous devons rechercher là où elle est définie, dans un autre ft, nous rappelons alors le jugement $\vdash_f$ avec l'environnement $\mathcal{F}$ supprimé du ft dans lequel y est défini. Et dans le cas où il n'existe pas une telle variable dans un FT de $\mathcal{F}$, alors nous soulevons l'erreur Ω (via *F2*).

Dans le cas où y est une racine de persistance, alors elle n'est pas de type séquence, donc nous devons l'encapsuler dans des crochets, pour créer la valeur séquence de singleton ; ce qui n'est pas le cas, dans le cas contraire, puisque y étant défini dans un FT, alors elle dénote une séquence.

La fonction pattern appliquée à la ligne ft d'un tableau de filtrage, réécrit le motif $\mathbb{C}$Duce correspondant, en considérant que y a pour type : `[<`s_0 `(`$a_1 = s_1$ `;` $\ldots$ `;`$a_k = s_k$`)>[`$R_1 \ldots R_n$`]*]`.

Si x_i est une variable liée à un attribut, alors le motif associé à x_i sera : a_i=x_i&t_i. Sinon, nous utiliserons les expressions régulières de motifs pour capturer x_i, et son

y : [<s_0 ($a_1 = s_1$; ... ;$a_k = s_k$)>[$R_1 \ldots R_n$]*] $R_1 \ldots R_n$ est un produit maximal si R_i ne représente pas une séquence $s_{i+k} = t_{i+k}\&R_i$ sinon $s_{i+k} = t_{i+k}\&[R_i]$ $i = 1 \ldots n$

---

pattern($FT(y|tag|k|(x_0,t_0)|(x_1,t_1)\ldots(x_k,t_k)|(x_{k+1},t_{k+1})\ldots(x_{k+n},t_{k+n}))) \rightsquigarrow$
<(x_0&t_0&s_0) a_1=x_1&t_1&s_1 ; ... ;a_k=x_k&t_k&s_k>[x_{k+1}::s_{k+1} ... x_{k+n}::s_{k+n}]

FIG. 6.19 – la fonction pattern()

motif associé sera : x_{i+k} :: s_{i+k}. Et s_{i+k} est calculé suivant la forme de R_i. Dans le cas, où R_i ne représente pas une séquence de types, comme par exemple le TRegexp `title`, alors $s_{i+k} = t_{i+k}\&R_i$ sinon (par exemple pour le TRegexp `book*`) $s_{i+k} = t_{i+k}\&[R_i]$.

bib	#	book*
doc	$(x_0, _)$	$(books, _)$

book	#	title	author+	publisher	price
books	$(x_1, _)$	$(titles, _)$	$(x_2, _)$	$(x_3, _)$	$(prices, _)$

<result>	title	price
q	*titles*	*prices*

FIG. 6.20 – Mettre le titre et le prix de chaque livre dans une élément `<result>`

Calculons la requête q en suivant l'algorithme de la figure 6.18. Nous avons les environnements :

- $\mathcal{C} = \{CT(q|$<result>$|0|\ |titles\ prices)\}$
- $\mathcal{F} = \{FT(doc|bib|0|(x_0, \texttt{Any})|\ |(books, \texttt{Any}))$
  $FT(books|book|0|(x_1, \texttt{Any})|(titles, \texttt{Any})(x_2, \texttt{Any})(x_3, \texttt{Any})(prices, \texttt{Any}))\}$

La règle R2 est évaluée en premier puisqu'il existe un CT dans $\mathcal{C}$ qui déclare la requête q. On a donc :

$\mathcal{C}, \mathcal{F} \vdash_s x \rightarrow$ `select <result> [ !titles !prices ] from` l_1,l_2

Le calcul de l_1 qui correspond à *titles* (qui est basée sur *books* qui n'est pas une racine de persistance), nous donne :

$\mathcal{F} \vdash_f titles \rightarrow l_3$,`<(x1)>[titles::title x2::author+ x3::publisher prices::price] in books`

On répète l'opération pour avoir l_3 sur *books* qui est basée sur *doc* qui est une racine de persistance :

$\mathcal{F} \vdash_f books \rightarrow$ `<(x0)> [ books::book* ] in [doc]`

On a donc que :

$l_1 =$ `<(x0)> [ books::book* ] in [doc],`
`<(x1)> [ titles::title x2::author+ x3::publisher prices::price] in books`

La même opération nous donne pour l_2 :

$l_2 =$ `<(x0)> [ books::book* ] in [doc],`

`<(x1)> [ titles::title x2::author+ x3::publisher prices::price] in books`

Les règles d'inférence présentées vont donc donner pour les tableaux de la figure 6.20 la requête $\mathbb{C}$QL suivante :

```
select <result> [ !titles !prices ]
from <(x0)> [ books::book* ] in [doc],
     <(x1)> [ titles::title x2::author+ x3::publisher prices::price]
                                                  in books,
     <(x0)> [ books::book* ] in [doc],
     <(x1)> [ titles::title x2::author+ x3::publisher prices::price]
                                                  in books
```

Comme on le voit, il y a des lignes de la clause `from` en trop. Cela est dû au fait que l'algorithme a dû chercher plusieurs fois les motifs permettant de définir les variables $titles$ et $prices$. Cela va donc générer de la redondance, et « doubler » (dans ce cas) les réponses. Pour éviter cela il faut se doter d'un nouvel environnement qui enregistre les variables définies précédemment par l'algorithme, que nous introduisons ci-après.

6.4.4.2 Cas de requêtes non-imbriquées et sans condition avec retrait de redondance

L'algorithme de la figure 6.21 permet de retirer cette redondance par l'utilisation d'un nouvel environnement Σ pour le jugement $\vdash_f$, qui va stocker les variables définies dans les motifs construits par la fonction pattern(). Nous rajoutons la règle *F1* qui renvoie un ensemble vide de « motif `in` expression », dans le cas, où la variable à chercher a déjà été vue, et donc appartient à Σ.

Nous modifions la règle *R2* avec l'expression $\mathcal{F}, \Sigma_{i-1} \vdash_f x_i \rightarrow (l_i, \Sigma_i)$ où les Σ_i servent d'accumulateurs et contiennent toutes les variables définies pour les Σ_k, tel que $k < i$ et $\Sigma_0 = \varnothing$. Les règles *F2*, *F3* et *F4* renvoient alors pour chaque environnement Σ_i un nouvel environnement qui ajoute les variables définies dans les l_i. Le dernier environnement Σ_n contiendra alors toutes les variables vues. Nous rajoutons la règle *F1* pour traiter le cas où la variable à chercher est déjà présente dans l'environnement Σ, dans ce cas, il n'y a aucune ligne l_i à rajouter dans la construction de la requête.

$$\frac{CT(x|tag|k|(a_1,x_1)\ldots(a_k,x_k)|x_{k+1}\ldots x_{k+n}) \in \mathcal{C} \qquad \mathcal{F}, \Sigma_{i-1} \vdash_f x_i \rightarrow (l_i, \Sigma_i) \quad \Sigma_0 = \varnothing \quad i = 1\ldots k+n}{\mathcal{F}, \mathcal{C} \vdash_s x \rightarrow \texttt{select <}tag\ a_1\texttt{=}x_1\ldots a_k\texttt{=}x_k\texttt{>[!}x_{k+1}\ldots\ \texttt{!}x_{k+n}\texttt{] from } l_1,\ldots,l_{k+n}} \ (R2)$$

$$\frac{\nexists ct \in \mathcal{C}, x \in FV(ct)}{\mathcal{F}, \mathcal{C} \vdash_s x \rightarrow \Omega} \ (R6)$$

$$\frac{x \in \Sigma}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (\varnothing, \Sigma)} \ (F1)$$

$$\frac{x \notin \Sigma \qquad \nexists ft \in \mathcal{F}, x \in DV(ft)}{\mathcal{F}, \Sigma \vdash_f x \rightarrow \Omega} \ (F2)$$

$$\frac{x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \quad y \in \mathcal{P} \quad \text{pattern}(ft) \rightsquigarrow p}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (p\ \texttt{in [}y\texttt{]}, \Sigma \cup DV(ft))} \ (F3)$$

$$\frac{x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \quad y \notin \mathcal{P} \quad \text{pattern}(ft) \rightsquigarrow p \qquad \mathcal{F}\backslash ft, \Sigma \cup DV(ft) \vdash_f y \rightarrow (l_i, \Sigma')}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (l_i\ \texttt{,}\ p\ \texttt{in}\ y, \Sigma')} \ (F4)$$

FIG. 6.21 – Règles d'inférence pour les requêtes PBE non imbriquées et sans boîte de condition avec le retrait de la redondance.

Avec cet algorithme la requête précédente nous donne la requête ℂQL suivante, qui correspond à ce qu'on attendait après avoir écrit ces tableaux.

```
select <result> [ !titles !prices ]
from <(x0)> [ books::book* ] in [doc],
     <(x1)> [ titles::title x2::author+ x3::publisher prices::price]
                                                        in books
```

6.4.4.3 Cas de requêtes imbriquées et sans condition

Nous étendons le précédent algorithme en ajoutant la possibilité d'avoir des requêtes imbriquées, c'est-à-dire, de pouvoir avoir dans des tableaux de construction des variables décrivant d'autres tableaux de construction.

Nous devons donc vérifier si une variable dans un tableau de construction est une variable définie dans un tableau de filtrage ou représente un tableau de construction. Pour cela, nous introduisons dans l'algorithme donné dans la figure 6.22, l'opérateur $\pi_{\mathcal{F}}\{Var\}$ qui nous donne le sous-ensemble de toutes les variables de Var déjà définies dans des FT. Pour chaque x_i dans la règle *R2* nous

$$\frac{\exists ft \in \mathcal{F}, x \in DV(ft)}{\mathcal{F}, \mathcal{C} \vdash_s x \rightarrow x} \ (R1)$$

$$\frac{\begin{array}{c} CT(x|tag|k|(a_1, x_1) \ldots (a_k, x_k)|x_{k+1} \ldots x_{k+n}) \in \mathcal{C} \quad \mathcal{F}, \mathcal{C} \vdash_s x_i \rightarrow Q_i \\ \{x_{j_1}, \ldots, x_{j_m}\} = \pi_{\mathcal{F}}(\{x_1, \ldots, x_{k+n}\}) \quad \mathcal{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \\ \Sigma_0 = \varnothing \quad i = 1 \ldots k+n \quad h = 1 \ldots m \end{array}}{\mathcal{F}, \mathcal{C} \vdash_s x \rightarrow \texttt{select <}tag\ a_1\texttt{=}Q_1 \ldots a_k\texttt{=}Q_k\texttt{>[!}Q_{k+1}\texttt{... !}Q_{k+n}\texttt{] from } l_1, \ldots, l_m} \ (R2)$$

$$\frac{\nexists ft \in \mathcal{F}, x \in DV(ft) \qquad \nexists ct \in \mathcal{C}, x \in DV(ct)}{\mathcal{F}, \mathcal{C} \vdash_s x \rightarrow \Omega} \ (R6)$$

$$\frac{x \in \Sigma}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (\varnothing, \Sigma)} \ (F1)$$

$$\frac{x \notin \Sigma \qquad \nexists ft \in \mathcal{F}, x \in DV(ft)}{\mathcal{F}, \Sigma \vdash_f x \rightarrow \Omega} \ (F2)$$

$$\frac{\begin{array}{c} x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \\ y \in \mathcal{P} \quad \text{pattern}(ft) \rightsquigarrow p \end{array}}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (p\ \texttt{in [}y\texttt{]}, \Sigma \cup DV(ft))} \ (F3)$$

$$\frac{\begin{array}{c} x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \\ y \notin \mathcal{P} \quad \text{pattern}(ft) \rightsquigarrow p \qquad \mathcal{F} \backslash ft, \Sigma \cup DV(ft) \vdash_f y \rightarrow (l_i, \Sigma') \end{array}}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (l_i\ \texttt{,}\ p\ \texttt{in}\ y, \Sigma')} \ (F4)$$

FIG. 6.22 – Règles d'inférence pour les requêtes imbriquées sans condition

appelons la règle *R1* qui renvoie la variable si elle est bien définie dans un tableau de filtrage ; sinon c'est alors une variable définissant un tableau de construction, on rappelle donc la règle *R2*, qui nous reconstruira une autre requête ℂQL. On suppose que chaque variable est définie une seule fois dans tous les tableaux. Donc la variable x de la règle *R2* ne peut être définie que dans un tableau de construction.

Étudions l'algorithme précédent appliqué aux tableaux de la figure 6.23 contenant des tableaux de construction imbriqués.

On a $\mathcal{C} = \{CT(q|\texttt{<result>}|0|\ |titles\ q')\ CT(q'|\texttt{<auteur>}|0|\ |ln\ fn)\}$.
On cherche à évaluer la requête q via *R2* : $\mathcal{F}, \mathcal{C} \vdash_s x$.
On a $\pi_{\mathcal{F}}(\{titles\ q'\}) = \{titles\}$ puisque $titles$ est la seule variable des deux à être définie dans un FT. On va aussi calculer $\mathcal{F}, \mathcal{C} \vdash_s titles$, et $\mathcal{F}, \mathcal{C} \vdash_s x'$.
Dans le premier cas, puisque $titles$ est définie dans un FT alors nous renvoyons l'expression ℂDuce : `titles` (via *R1*). Nous évaluons aussi l'ensemble de la clause `from`, ce qui nous donne un environnement Σ_1 de variables et un ensemble l_1 de

bib	#	book*
doc	$(x_0, _)$	$(books, _)$

book	#	title	author+	publisher	price
books	$(x_1, _)$	$(titles, _)$	$(a, _)$	$(x_2, _)$	$(x_3, _)$

author	#	last	first
a	$(x_4, _)$	$(ln, _)$	$(fn, _)$

<result>	title	<auteur>[last first]
q	*titles*	q'

<auteur>	last	first
q'	**ln**	**fn**

FIG. 6.23 – Mettre dans un élément <result> les titres et les auteurs, en changeant la balise author par la balise auteur.

lignes de la clause from.

Nous obtenons donc pour cette première étape :

$\mathcal{F}, \mathcal{C} \vdash_s x \rightarrow$ select <result>[!titles $!Q_2$]

from <(x0)>[books:: book*] in [doc],

<(x1)>[titles::title a:: author+ x2::publisher x3::price] in books où Q_2 va correspondre à l'évaluation de q'.

Dans le deuxième cas, puisque q' est une variable définie dans un CT, alors nous réévaluons la règle *R2* sur q'. Puisque le CT définissant q' ne contient que des variables définies dans des FT, alors l'algorithme se déroule comme précédemment.

On obtient donc au final la requête $\mathbb{C}$QL suivante :

```
select <result>[!titles
      !select <auteur>[ !ln !fn]
       from  <(x0)>[books::book*] in [doc],
             <(x1)>[titles::title a::author+ x2::publisher x3::price]
                                                    in books
             <(x4)>[ln::last fn::first] in a
      ]
from <(x0)>[books:: book*  ] in [doc],
     <(x1)>[titles::title a:: author+  x2::publisher x3::price ]
                                                    in books
```

Nous voyons que nous avons le même problème de redondance que précédemment, qui est la répétition de motifs pour capturer des variables qui sont utilisées dans des tableaux de construction différents. Nous allons donc rajouter un nouvel environnement pour stocker les variables définies dans chaque tableaux de construction.

6.4.4.4 Cas de requêtes imbriquées et sans condition avec retrait de redondance

$$\frac{\exists ft \in \mathcal{F}, x \in DV(ft)}{\mathcal{F}, \mathcal{C}, \Sigma \vdash_s x \rightarrow (\Sigma, x)} \ (R1)$$

$$\frac{\begin{array}{l} CT(x|tag|k|(a_1,x_1)..(a_k,x_k)|x_{k+1}..x_{k+n}) \in \mathcal{C} \\ \{x_{j_1},\ldots,x_{j_m}\} = \pi_{\mathcal{F}}(\{x_1,\ldots,x_{k+n}\}) \\ \mathcal{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \quad \mathcal{F}, \mathcal{C}, \Sigma_m \vdash_s x_i \rightarrow (\Sigma'_i, Q_i) \\ h = 1 \ldots m \quad i = 1 \ldots k+n \end{array}}{\mathcal{F}, \mathcal{C}, \Sigma_0 \vdash_s x \rightarrow (\Sigma_m, \texttt{select <}tag\ a_1\texttt{=}Q_1..a_k\texttt{=}Q_k\texttt{>[!}Q_{k+1}..\texttt{ !}Q_{k+n}\texttt{] from } l_1..l_m)} \ (R2)$$

$$\frac{\nexists ft \in \mathcal{F}, x \in DV(ft) \qquad \nexists ct \in \mathcal{C}, x \in FV(ct)}{\mathcal{F}, \mathcal{C}, \Sigma \vdash_s x \rightarrow \Omega} \ (R6)$$

$$\frac{x \in \Sigma}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (\varnothing, \Sigma)} \ (F1)$$

$$\frac{x \notin \Sigma \qquad \nexists ft \in \mathcal{F}, x \in DV(ft)}{\mathcal{F}, \Sigma \vdash_f x \rightarrow \Omega} \ (F2)$$

$$\frac{\begin{array}{l} x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \\ y \in \mathcal{P} \quad \textsf{pattern}(ft) \rightsquigarrow p \end{array}}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (p\ \texttt{in [}y\texttt{]}, \Sigma \cup DV(ft))} \ (F3)$$

$$\frac{\begin{array}{l} x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \\ y \notin \mathcal{P} \quad \textsf{pattern}(ft) \rightsquigarrow p \qquad \mathcal{F} \backslash ft, \Sigma \cup DV(ft) \vdash_f y \rightarrow (l_i, \Sigma') \end{array}}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (l_i\ \texttt{,}\ p\ \texttt{in}\ y, \Sigma')} \ (F4)$$

FIG. 6.24 – Règles d'inférence pour les requêtes imbriquées sans condition avec retrait de la redondance.

Les règles d'inférence sont données dans la figure 6.24 avec l'ajout d'un nouvel environnement Σ dans le jugement $\vdash_s$ qui va stocker toutes les variables définies dans un tableau de construction pour éviter de les redéfinir dans un autre CT.

À partir des tableaux de la figure 6.23, nous aurons donc la requête suivante :

```
select <result>[!titles
      !select <auteur>[ !ln !fn]
       from  <(x4)>[ln::last fn::first] in a
       ]
from <(x0)>[books:: book*  ] in [doc],
     <(x1)>[titles::title a:: author+  x2::publisher x3::price ] in books
```

Il existe néanmoins un problème dans le cas où dans la règle *R2*, on a que $\Sigma_0 = \Sigma_m$, c'est-à-dire le cas où une clause `from` ne contiendrait aucun motif, car toutes les variables auront déjà été déclarées précédemment, ce qui lèverait une erreur de syntaxe (car une requête avec une clause `from` vide n'est pas une requête $\mathbb{C}$QL).

Un exemple de ce genre de cas est donné dans la figure 6.25, où toutes les variables nécessaires à la définition de $titles$ (à savoir doc et $books$) sont les mêmes que celles pour a.

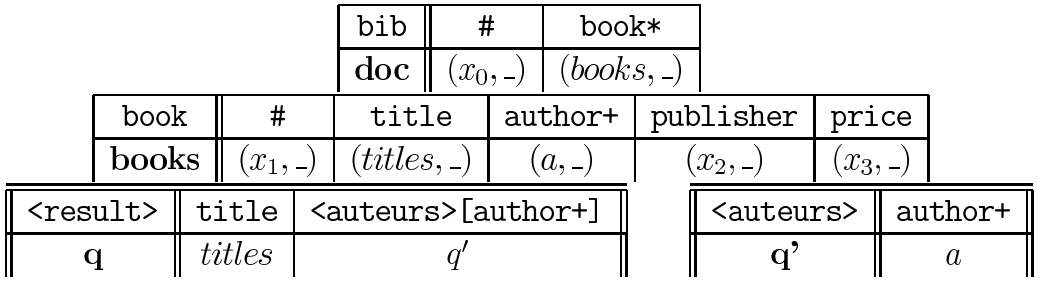


FIG. 6.25 – Exemple de requête imbriquée telle que l'ensemble des variables à déclarer de l'une soit le même que dans la seconde.

Avec les règles d'inférence de la figure 6.24, nous obtiendrions la requête suivante :

```
select <result>[!titles
      !select <auteurs>[ !a]
       from
       ]
from <(x0)>[books:: book*  ] in [doc],
     <(x1)>[titles::title a:: author+  x2::publisher x3::price ]
                                                        in books
```

Cette requête n'a pas la bonne syntaxe puisque la clause `from` est vide. Pour éviter ce problème, nous proposons à la place du `select` avec une clause `from` vide, l'expression `[e]` qui est le résultat souhaité. Nous rajoutons donc une nouvelle règle R4 dans le cas où $\Sigma_0 = \Sigma_m$ (et modifions la règle *R2* avec la condition $\Sigma_0 \neq \Sigma_m$) :

$$\frac{\begin{array}{l}(\text{si } \Sigma_0 = \Sigma_m)\\ CT(x|tag|k|(a_1,x_1)\ldots(a_k,x_k)|x_{k+1}\ldots x_{k+n}) \in \mathcal{C}\\ \{x_{j_1},\ldots,x_{j_m}\} = \pi_{\mathcal{F}}(\{x_1,\ldots,x_{k+n}\})\\ \mathcal{F},\Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h,\Sigma_h) \quad \mathcal{F},\mathcal{C},\Sigma_m \vdash_s x_i \rightarrow (\Sigma'_i,Q_i)\\ i = 1\ldots k+n \quad h = 1\ldots m\end{array}}{\mathcal{F},\mathcal{C},\Sigma_0 \vdash_s x \rightarrow (\Sigma_m, \texttt{[<}tag\ \ a_1\texttt{=}Q_1\ldots a_k\texttt{=}Q_k\texttt{>[!}Q_{k+1}\ldots\ \texttt{!}Q_{k+n}\texttt{]]})}\ (R4)$$

Ce qui fait qu'avec les tableaux de la figure 6.25, nous obtiendrons la requête suivante :

```
select <result>[!titles ![<auteurs>[!a]]]
from <(x0)>[books:: book*  ] in [doc],
     <(x1)>[titles::title a:: author+  x2::publisher x3::price ] in books
```

6.4.4.5 Cas de requêtes imbriquées avec condition

Nous introduisons dans les figures 6.26, 6.27 et 6.28 un nouvel environnement Θ qui contient toutes les lignes de la boîte de condition. Nous devons considérer les conditions avec une variable, et celles avec deux variables.

Dans le cas d'une condition avec une variable, nous avons juste à la placer dans la clause `where`, dès que nous définissons cette variable dans une clause `from`.

Dans le cas où nous avons plus de variables, nous devons attendre que toutes les variables soient définies pour placer la condition. Il se peut qu'il reste des conditions qui contiennent une variable définie et qu'une autre n'ait pas été définie. Nous pensons qu'il est alors important de les considérer, ne serait-ce que dans le cas d'une condition de jointure.

Dans ce cas nous avons alors deux stratégies de traduction :

- Soit dès qu'une variable définie dans un FT porte sur une variable d'un condition CB, et dans ce cas, nous allons "chercher" la variable manquante dans la condition, pour pouvoir la définir dans le tableau de filtrage.
- Soit nous attendons que toutes les variables d'une condition soient définies dans un motif, pour alors placer la condition dans une clause `where`. Et s'il reste des conditions telles qu'une des ses variables a été définie mais pas l'autre, nous relançons alors l'algorithme pour trouver ces variables dans des FT.

Nous n'avons utilisé que la première stratégie.

Dans les règles *R2* et *R3*, nous étudions le cas où il existe des conditions qui peuvent être évaluées avec des variables déjà définies et qui sont stockées dans

$$\frac{\exists ft \in \mathcal{F}, x \in DV(ft)}{\mathcal{F}, \mathcal{C}, \Sigma \vdash_s x \rightarrow (\Sigma, x)} \ (R1)$$

(si $\Sigma_0 \neq \Sigma_m$)
$CT(x|tag|k|(a_1, x_1) \ldots (a_k, x_k)|x_{k+1} \ldots x_{k+n}) \in \mathcal{C} \quad \{x_{j_1}, \ldots, x_{j_m}\} = \pi_{\mathcal{F}}(\{x_1, \ldots, x_{k+n}\})$
$\mathcal{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \quad \mathcal{F}, \mathcal{C}, \Sigma_m, \Theta \vdash_s x_i \rightarrow (\Sigma'_i, Q_i)$
$\mathcal{F}, \Sigma_m, \Theta \vdash_c (\varnothing, \varnothing, \Sigma_m) \quad i = 1 \ldots k+n \quad h = 1 \ldots m$

$$\frac{}{\mathcal{F}, \mathcal{C}, \Sigma_0, \Theta \vdash_s x \rightarrow (\Sigma_m, \texttt{select <}tag\ a_1\texttt{=}Q_1 \ldots a_k\texttt{=}Q_k\texttt{>[!}Q_{k+1} \ldots \texttt{ !}Q_{k+n}\texttt{] from } l_1, \ldots, l_m)} \ (R2)$$

(si $\Sigma_0 \neq \Sigma_m$)
$CT(x|tag|k|(a_1, x_1) \ldots (a_k, x_k)|x_{k+1} \ldots x_{k+n}) \in \mathcal{C} \quad \{x_{j_1}, \ldots, x_{j_m}\} = \pi_{\mathcal{F}}(\{x_1, \ldots, x_{k+n}\})$
$\mathcal{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \quad \mathcal{F}, \mathcal{C}, \Sigma_m, \Theta \vdash_s x_i \rightarrow (\Sigma'_i, Q_i)$
$\mathcal{F}, \Sigma_m, \Theta \vdash_c (C, l_c, \Sigma') \quad i = 1 \ldots k+n \quad h = 1 \ldots m$

$$\frac{}{\mathcal{F}, \mathcal{C}, \Sigma_0, \Theta \vdash_s x \rightarrow (\Sigma', \texttt{select <}tag\ a_1\texttt{=}Q_1\texttt{..}a_k\texttt{=}Q_k\texttt{>[!}Q_{k+1}\texttt{.. !}Q_{k+n}\texttt{] from } l_1\texttt{..}l_m\texttt{,}l_c \texttt{ where } C)} \ (R3)$$

(si $\Sigma_0 = \Sigma_m$)
$CT(x|tag|k|(a_1, x_1) \ldots (a_k, x_k)|x_{k+1} \ldots x_{k+n}) \in \mathcal{C} \quad \{x_{j_1}, \ldots, x_{j_m}\} = \pi_{\mathcal{F}}(\{x_1, \ldots, x_{k+n}\})$
$\mathcal{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \quad \mathcal{F}, \mathcal{C}, \Sigma_m, \Theta \vdash_s x_i \rightarrow (\Sigma'_i, Q_i)$
$\mathcal{F}, \Sigma_m, \Theta \vdash_c (\varnothing, \varnothing, \Sigma_m) \quad i = 1 \ldots k+n \quad h = 1 \ldots m$

$$\frac{}{\mathcal{F}, \mathcal{C}, \Sigma_0, \Theta \vdash_s x \rightarrow (\Sigma_m, \texttt{[<}tag\ a_1\texttt{=}Q_1 \ldots a_k\texttt{=}Q_k\texttt{>[!}Q_{k+1} \ldots \texttt{ !}Q_{k+n}\texttt{]]})} \ (R4)$$

(si $\Sigma_0 = \Sigma_m$)
$CT(x|tag|k|(a_1, x_1) \ldots (a_k, x_k)|x_{k+1} \ldots x_{k+n}) \in \mathcal{C} \quad \{x_{j_1}, \ldots, x_{j_m}\} = \pi_{\mathcal{F}}(\{x_1, \ldots, x_{k+n}\})$
$\mathcal{F}, \Sigma_{h-1} \vdash_f x_{j_h} \rightarrow (l_h, \Sigma_h) \quad \mathcal{F}, \mathcal{C}, \Sigma_m, \Theta \vdash_s x_i \rightarrow (\Sigma'_i, Q_i) \quad \mathcal{F}, \Sigma_m, \Theta \vdash_c (C, \varnothing, \Sigma_m)$
$i = 1 \ldots k+n \quad h = 1 \ldots m$

$$\frac{}{\mathcal{F}, \mathcal{C}, \Sigma_0, \Theta \vdash_s x \rightarrow (\Sigma_m, \texttt{if } C \texttt{ then [<}tag\ a_1\texttt{=}Q_1 \ldots a_k\texttt{=}Q_k\texttt{>[!}Q_{k+1} \ldots \texttt{ !}Q_{k+n}\texttt{]]) else []}} \ (R5)$$

$$\frac{\nexists ft \in \mathcal{F}, x \in DV(ft) \qquad \nexists ct \in \mathcal{C}, x \in FV(ct)}{\mathcal{F}, \mathcal{C}, \Sigma, \Theta \vdash_s x \rightarrow \Omega} \ (R6)$$

FIG. 6.26 – Règles d'inférence pour les requêtes imbriquées avec condition (partie I).

$$\frac{x \in \Sigma}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (\varnothing, \Sigma)}\ (F1) \qquad \frac{x \notin \Sigma \quad \nexists ft \in \mathcal{F}, x \in DV(ft)}{\mathcal{F}, \Sigma \vdash_f x \rightarrow \Omega}\ (F2)$$

$$\frac{\begin{array}{l} x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \\ y \in \mathcal{P} \quad \text{pattern}(ft) \rightsquigarrow p \end{array}}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (p \ \texttt{in}\ [y], \Sigma \cup DV(ft))}\ (F3)$$

$$\frac{\begin{array}{l} x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \\ y \notin \mathcal{P} \quad \text{pattern}(ft) \rightsquigarrow p \qquad \mathcal{F} \backslash ft, \Sigma \cup DV(ft) \vdash_f y \rightarrow (l_i, \Sigma') \end{array}}{\mathcal{F}, \Sigma \vdash_f x \rightarrow (l_i \ , \ p \ \texttt{in}\ y, \Sigma')}\ (F4)$$

FIG. 6.27 – Règles d'inférence pour les requêtes imbriquées avec condition (partie II).

l'environnement Σ. Dans le cas où il n'y en a pas nous n'écrivons pas de clause `where`. Dans le cas où il y en a, nous allons calculer l'expression de la clause `where` avec le jugement $\vdash_c$.

Les règles *R4* et *R5* remplacent la règle précédente *R4* de la figure 6.24, c'est-à-dire dans le cas où l'appel du jugement $\vdash_f$ ne retourne aucune ligne pour la clause `from`.

Dans le cas où le jugement $\vdash_c$ renvoie une clause `where` alors nous appliquons la règle *R3*.

Dans le cas éventuel où il y a une condition à retourner mais pas de motifs à déclarer dans la cause `from`, alors nous décidons de renvoyer une expression `if-then-else`.

Le jugement $\vdash_c$ est défini dans la figure 6.28 tel que :

$\mathcal{F}, \Sigma, \Theta \vdash_c (C, l, \Sigma')$

où $\mathcal{F}$ est l'environnement contenant tous les FT, Σ l'environnement contenant toutes les variables déjà vues et définies dans des motifs ; et Θ est l'environnement contenant toutes les CB et renvoie un triplet avec une expression C à poser dans la clause `where`, une expression l contenant toutes les nouvelles lignes à rajouter dans la clause `from` et correspondant aux motifs qui ont été nécessaires pour définir les variables d'une condition, et le nouvel environnement Σ' (modifié uniquement par l'appel de la règle *C7*) qui contiendra toutes ces nouvelles variables définies dans l.

L'exemple de la figure 6.29 permet d'exprimer une jointure avec une condition de jointure contenant deux variables (il s'agit de la requête Q5 des « *XML Query Use Cases* » [CFF+03]). C'est un exemple montrant qu'on a besoin d'aller définir

$$\frac{}{\mathcal{F}, \varnothing, \Theta \vdash_c (\varnothing, \varnothing, \varnothing)}\ (C1) \qquad \frac{}{\mathcal{F}, \Sigma, \varnothing \vdash_c (\varnothing, \varnothing, \Sigma)}\ (C2)$$

$$\frac{cb = CB(op, x, v) \in \Theta \quad x \notin \Sigma \quad \mathcal{F}, \Sigma, \Theta \backslash cb \vdash_c (C, l, \Sigma')}{\mathcal{F}, \Sigma, \Theta \vdash_c (C, l, \Sigma')}\ (C3)$$

$$\frac{cb = CB(op, x, v) \in \Theta \quad x \in \Sigma \quad \mathcal{F}, \Sigma, \Theta \backslash cb \vdash_c (C, l, \Sigma')}{\mathcal{F}, \Sigma, \Theta \vdash_c (C \text{ and } (x\ op\ v), l, \Sigma')}\ (C4)$$

$$\frac{cb = CB(op, x_1, x_2) \in \Theta \quad x_1 \notin \Sigma \quad x_2 \notin \Sigma \quad \mathcal{F}, \Sigma, \Theta \backslash cb \vdash_c (C, l, \Sigma')}{\mathcal{F}, \Sigma, \Theta \vdash_c (C, l, \Sigma')}\ (C5)$$

$$\frac{cb = CB(op, x_1, x_2) \in \Theta \quad x_1 \in \Sigma \quad x_2 \in \Sigma \quad \mathcal{F}, \Sigma, \Theta \backslash cb \vdash_c (C, l, \Sigma')}{\mathcal{F}, \Sigma, \Theta \vdash_c (C \text{ and } (x_1\ op\ x_2), l, \Sigma')}\ (C6)$$

$$\frac{\begin{array}{l} cb = CB(op, x_1, x_2) \in \Theta \quad x_1 \in \Sigma \quad x_2 \notin \Sigma \quad \mathcal{F}, \Sigma, \Theta \backslash cb \vdash_c (C, l_1, \Sigma') \\ \mathcal{F}, \Sigma' \vdash_f x_2 \rightarrow (l_2, \Sigma'') \end{array}}{\mathcal{F}, \Sigma, \Theta \vdash_c (C \text{ and } (x_1\ op\ x_2), l_1, l_2, \Sigma'')}\ (C7)$$

FIG. 6.28 – Règles d'inférence pour les requêtes imbriquées avec condition (partie III).

toutes les variables apparaissant dans une boîte de condition si l'une d'entre elles apparaît. Ici, pour calculer q, on a besoin de définir $title1$, qui est définie dans le tableau de filtrage sur $books$. Et $books$ est défini par le tableau de filtrage sur la racine de persistance doc. Nous avons donc besoin d'aller chercher la définition de $title2$, sur $entry$, et donc $bstore2$.

Sur application de ces règles nous obtenons la requête ℂQL suivante :

```
select <result>[!title1
from <(x0)>[books:: book*  ] in [doc],
     <(x1)>[title1::title x2 ::author+ x3::publisher x4::price ]
                                    in books,
     <(x5)>[reviews:: entry*  ] in [bstore2]
     <(x6)>[title2::title x7::price x8::review ] in reviews
where title1=title2
```

6.4.4.6 Recherche descendante de tableaux de filtrage.

L'algorithme précédent construit une requête à partir des variables données dans un tableau de construction, et recherche toutes les variables nécessaires à cette construction dans les autres tableaux de filtrage. Mais peut-être que l'utilisateur a

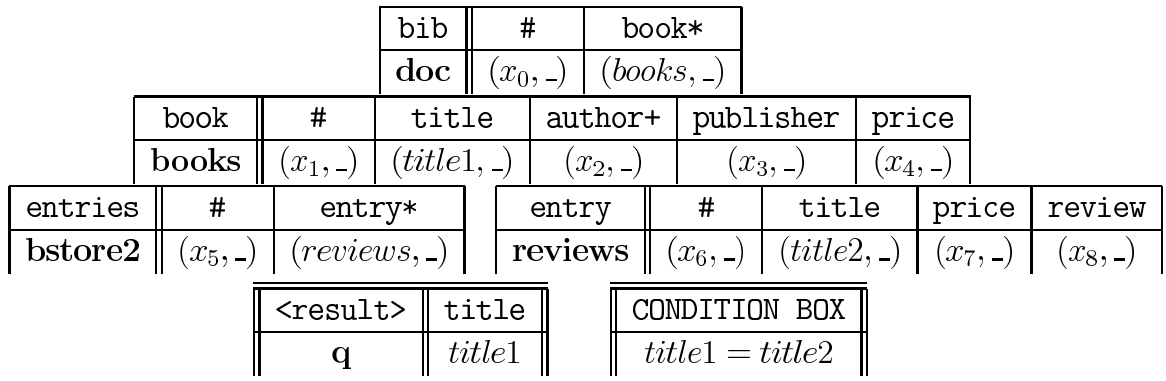


FIG. 6.29 – Quels sont les titres qui apparaissent dans la bibliographie et dans `bstore2` ?

pu donner des contraintes (à l'aide d'expression régulières de types) dans d'autres tableaux de filtrage qui portent sur des variables qui se trouvent dans les tableaux nécessaires à la construction de la requête. Nous décidons alors d'en tenir compte pour la création de la requête, et donc de « descendre » dans les tableaux de filtrage pour trouver tous ceux qui portent sur une variable déjà définie.

Un exemple de ce genre de tableaux est donné dans la figure 6.30. Dans cet exemple, nous aimerions pouvoir avoir la condition sur la variable $prices$ donnée dans le tableau de filtrage sur $prices$, dans l'écriture de la requête correspondant à q.
Il s'agit d'un choix de conception pour ce langage. Nous pensons qu'il peut être utile que des tableaux de filtrage puissent influencer d'autres tableaux de filtrage. Nous permettons ainsi l'utilisation de tableaux de filtrage pour poser des conditions à des endroits de l'arbre XML qui n'auront pas été directement explorés pour capturer les sous-abres nécessaires pour construire une requête.

Nous augmentons donc les règles d'inférence précédentes en ajoutant la règle *FD4* (qui utilise le jugement $\vdash_{fd}$), pour étudier les éventuels tableaux qui utilisent une variable définie, et qui pourraient apporter des conditions supplémentaires. Ces règles sont données dans les figures ?? et 6.31 (ainsi que les règles pour les conditions données dans la figure 6.28, et celles de la clause `select` données dans la figure 6.26).

Cet algorithme nous permettra d'avoir la requête $\mathbb{C}$QL suivante qui donc contient le motif qui porte sur les fils de `prices`.

```
select <result>[!title1]
from <(x0)>[books:: book*] in [doc],
     <(x1)>[title1::title x2::author+ x3::publisher prices::price] in books,
     <(x4)>[x5 & 50--*] in prices
```

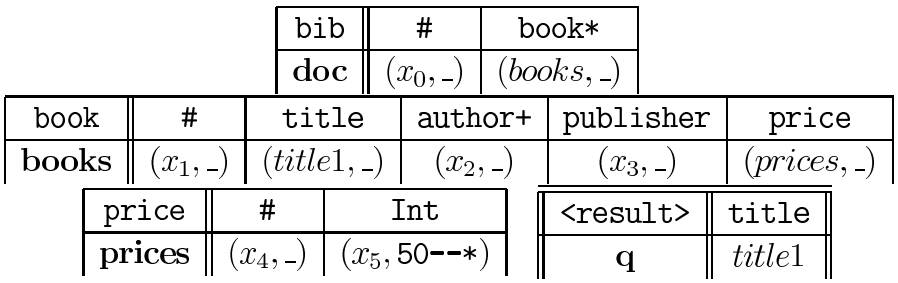


FIG. 6.30 – Quels sont les titres des livres dont le prix est supérieur à 50.

6.4.4.7 Union et intersection dans les tableaux

Nous avons donné une sémantique sur les lignes permettant de poser une disjonction sur les lignes d'un tableau de filtrage, comme présenté dans la visite guidée dans la figure 6.15 à la page 138. Il suffit de détecter la présence de deux FT portant sur la même variable et étant définies dans le même tableau. Dans ce cas, si p_1 est le motif associé à la première ligne l_1 et p_2 le motif associé à une autre ligne l_2, et y la variable sur lesquelles portent ces lignes, alors nous ne renvoyons qu'une seule ligne dans la clause `from`, qui est l'alternative sur ces deux motifs : $p_1|p_2$ `in` y. Ce qui est rendu possible, car toutes les variables sont identiques pour chaque colonne d'un tableau, d'après la définition des tableaux de filtrage. Il faut aussi vérifier que l_1 et l_2 appartiennent au même tableau.

Dans le cas contraire, où l_1 et l_2 n'appartiennent pas au même tableau, nous ne considérerons alors pas que ces deux lignes représentent une disjonction. Dans ce cas, l'algorithme produira deux lignes de la clause `from` différentes (p_1 `in` y, ..., p_2 `in` y), ce qui correspond d'une certaine manière à une conjonction de ces deux motifs ; si un des deux n'est pas filtré alors l'évaluation s'arrête.

6.5 Conclusion

En conclusion nous avons présenté dans ce chapitre notre outil visuel graphique PBE qui a été conçu pour permettre d'exprimer des requêtes $\mathbb{C}$QL sans connaissance du langage $\mathbb{C}$Duce et de l'expressivité des motifs, et qui s'inspire fortement de QBE. L'écriture de tableaux se fait en deux étapes, l'une pour définir les éléments à capturer l'autre pour reconstruire des requêtes de manière à éventuellement les composer. Un utilisateur peu averti pourra construire des requêtes complexes grâce à l'algorithme de traduction présenté, et qui prend en compte toutes les possibilités dont un utilisateur dispose pour exprimer à l'aide de tableaux des requêtes $\mathbb{C}$QL. Cet algorithme n'utilise que des motifs décrits dans la DTD donnée, et n'a donc

$$\frac{x \in \Sigma}{\mathcal{F}, \Sigma \vdash_f x \to (\varnothing, \Sigma)}\ (\textit{F1}) \qquad \frac{x \notin \Sigma \quad \nexists ft \in \mathcal{F}, x \in DV(ft)}{\mathcal{F}, \Sigma \vdash_f x \to \Omega}\ (\textit{F2})$$

$$\frac{\begin{array}{l} x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \\ y \in \mathcal{P} \quad \textsf{pattern}(ft) \rightsquigarrow p \quad \{x_1 \ldots x_n\} = DV(ft) \quad \Sigma_0 = \Sigma \cup DV(ft) \\ \mathcal{F} \backslash ft, \Sigma_{i-1} \vdash_{fd} x_i \to (l_i, \Sigma_i) \quad i = 1 \ldots n \end{array}}{\mathcal{F}, \Sigma \vdash_f x \to (p \texttt{ in [}y\texttt{],} l_i, \Sigma_n)}\ (\textit{F3})$$

$$\frac{\begin{array}{l} x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in DV(ft) \quad y \in FV(ft) \quad \{x_1 \ldots x_n\} = DV(ft) \\ y \notin \mathcal{P} \quad \textsf{pattern}(ft) \rightsquigarrow p \qquad \mathcal{F} \backslash ft, \Sigma \cup DV(ft) \vdash_f y \to (l_y, \Sigma_0) \\ \mathcal{F} \backslash ft, \Sigma_{i-1} \vdash_{fd} x_i \to (l_i, \Sigma_i) \quad i = 1 \ldots n \end{array}}{\mathcal{F}, \Sigma \vdash_f x \to (l_y \texttt{ , } p \texttt{ in } y \texttt{ , } l_i, \Sigma_n)}\ (\textit{F4})$$

$$\frac{x \in \Sigma}{\mathcal{F}, \Sigma \vdash_{fd} x \to (\varnothing, \Sigma)}\ (\textit{FD1}) \qquad \frac{x \notin \Sigma \quad \nexists ft \in \mathcal{F}, x \in FV(ft)}{\mathcal{F}, \Sigma \vdash_{fd} x \to (\varnothing, \Sigma)}\ (\textit{FD2})$$

$$\frac{\begin{array}{l} x \notin \Sigma \quad \exists ft \in \mathcal{F}, x \in FV(ft) \\ \quad \textsf{pattern}(ft) \rightsquigarrow p \quad \{x_1 \ldots x_n\} = DV(ft) \\ \quad \Sigma_0 = \Sigma \cup DV(ft) \mathcal{F} \backslash ft, \Sigma_{i-1} \vdash_{fd} x_i \to (l_i, \Sigma_i) \quad i = 1 \ldots n \end{array}}{\mathcal{F}, \Sigma \vdash_{fd} x \to (p \texttt{ in } y\texttt{,} l_i, \Sigma_n)}\ (\textit{FD4})$$

FIG. 6.31 – Règles d'inférence pour les requêtes imbriquées avec condition et recherche descendante.

pas besoin d'un document XML si la DTD est suffisamment expressive. Comme PBE ne construira que de requêtes ne possédant que le déconstructeur de filtrage par motifs, ces requêtes seront alors performantes comme on a pu le voir dans le chapitre 5.

PBE a été développé en OCaml à l'aide de l'outil graphique GTK via LablGTK, et de ℂDuce pour le système de types. Il n'est pas encore distribué à l'heure actuelle, mais il le sera quand il possédera une interface esthétiquement plus aboutie, et possédera plus de fonctionnalités. Comme éventuelles améliorations, nous pourrions considérer la possibilité d'écrire ses propres DTD via des tableaux, retirer la nécessité de passer par des déclarations de variables et proposer uniquement une interface visuelle qui permette de capturer par des clics de souris les différents sous-arbres, et les replacer par une opération de « drag and drop » dans la partie de reconstruction. Nous étudierons la manière de représenter graphiquement des unions d'expressions régulières dans les tableaux de filtrage. Le but principal à atteindre serait de pouvoir exprimer des conditions sur chaque partie de l'union en évitant à l'utilisateur de recourir à des expressions régulières de motifs complexes.

Dans la suite nous essayerons de coupler PBE avec un moteur de requêtes XQuery, et de générer ainsi des requêtes XQuery. Cela pose des problèmes notamment à cause de l'utilisation de l'ordre des sous-éléments qui est sous-jacent à PBE. Par exemple, si nous avons un type `A` qui est égal à `<a>[B* C* B*]`, et que nous voulons capturer tous les éléments B en mettant une variable dans la première case, il faut alors générer une projection XPath équivalente. La requête `$a/B` ne retournera pas la même chose que le motif associé en ℂQL dans le cas ou l'élément `a` de type `A` contienne des éléments C, car en ℂQL on capturera tous les éléments `B` à gauche des `C`. Il sera alors intéressant dans la suite de répertorier toutes ces difficultés, et de proposer un algorithme de traduction en XQuery/XPath.

Chapitre 7

Conclusion et perspectives

7.1 Conclusion

Nous avons, tout le long de ce manuscrit, présenté les travaux qui se sont inscrits dans le cadre de cette thèse. L'objectif premier était d'inclure dans le langage ℂDuce un langage de requêtes offrant une interface véritablement déclarative.

L'enjeu étant de se reposer sur le mécanisme de filtrage comme primitive de déconstruction. Nous avons proposé la possibilité de combiner ce déconstructeur par filtrage par motifs avec des expressions descendantes de chemins (projections) à la XPath utilisées par XQuery. Ceci à donné lieu à la définition et l'implantation du langage ℂQL. Nous avons présenté sa syntaxe et sa sémantique dans le chapitre 3. Nous avons donné à travers différents exemples les différents intérêts complémentaires de l'utilisation séparée du filtrage par motifs, et des projections, et de l'utilisation combinée des deux. Ce langage de requête a été intégré dans ℂDuce et fait désormais partie de ses différentes distributions.

Partant du constat que le filtrage par motifs est plus performant car permettant de capturer différents sous-arbres en un seul motif, là où plusieurs projections sont nécessaires, nous avons apporté dans le chapitre 4 un algorithme de traduction d'une requête comportant des projections en une requête composée uniquement de motifs. Nous avons prouvé que cette traduction préserve la sémantique et le typage. Nous avons aussi apporté d'autres optimisations qui peuvent se combiner à cette traduction pour réduire le nombre de motifs. Nous avons aussi adapté des optimisations logiques classiques dans le cadre des bases de données (la descente des sélections), et qui ici fait remonter les sélections pour être placées sur les motifs sur lesquels elles portent.

Dans le chapitre 5 nous avons étudié les performances de ℂQL en suivant différentes approches. Dans un premier temps nous avons suivi l'approche tradi-

tionnelle des jeux de tests en se comparant à des moteurs de requêtes XQuery, comme Galax et Qizx. Il a été montré que l'utilisation du filtrage par motifs est toujours comparable et souvent meilleure que l'utilisation de projections, et que les requêtes avec projections traduites par l'algorithme du chapitre précédent permettent un gain de temps notable. Mais il a été montré aussi que l'utilisation de motifs ad-hoc écrits par un utilisateur averti permet de gagner du temps par rapport à ces requêtes traduites. L'optimisation de descente de sélections permet dans des requêtes sélectives de gagner du temps sur les requêtes avec motifs, ce qui prouve que les techniques classiques de bases de données sont pertinentes et adaptées. Et donc que les autres techniques d'optimisations mériteraient d'être étudiées dans ce cadre. En général ℂQL a des temps d'exécution comparables aux meilleures implantations de XQuery.
La deuxième approche est celle, nouvelle, dite des "microbenchmarks" et qui consiste à comparer des requêtes atomiques sur certains aspects, de manière à détecter les aspects performants de ceux qui le sont moins, dans le but d'améliorer ces derniers. Il a été montré que ℂQL se comporte bien comparé aux autres moteurs de requêtes, mais que certains points devraient être améliorés, comme le problème de mémoire pour les requêtes produisant des résultats volumineux.

Mettant en évidence que les requêtes écrites avec des motifs précis inspirés de la DTD sont plus performantes, mais qu'ils nécessitent une bonne compréhension du mécanisme du filtrage par motifs, ce qui n'est pas évident pour un utilisateur non averti, nous avons proposé une méthode d'aide à l'écriture de ces motifs, et apporté un outil graphique d'écriture de requêtes. Cet outil s'inspire du QBE et est appelé PBE pour « Pattern by Example ». Dans le chapitre 6 une visite guidée est donnée qui montre l'utilisation des tableaux de filtrage et de construction, suivie de la syntaxe et de la sémantique de ces tableaux, exprimée par le biais d'une réécriture de requêtes PBE en requête ℂQL.

7.2 Perspectives

Nous avons vu que certaines optimisations classiques pourraient être adaptées au cas de ℂQL comme celles basées sur les jointures qu'utilise Qizx par exemple. Cependant, nous pensons que les optimisations de ce type ne doivent pas être faites au niveau du langage de requêtes mais doivent être déléguées à un optimiseur. Il sera intéressant de coupler ℂQL avec un système de stockage de base de données, notamment dans le cadre des XAM « XML Acces Modules »[ABM05, ABMV05], et de proposer ainsi un optimiseur pour ℂQL. Cela résoudrait aussi le point soulevé

par l'étude de « microbenchmarks », sur le manque de sérialisation des données par ℂQL.

Il serait aussi intéressant d'adapter à ℂQL les techniques de [MS03, BCCN06] qui utilisent le typage de manière à n'explorer que les parties du document susceptibles d'être parcourues, et d'ainsi de minimiser les tailles des données intermédiaires.

Le travail de [Ngu04] permettra vraisemblablement d'approximer le type de la projection sur descendants, et donc d'enrichir ℂQL ; on pourra alors éventuellement étudier l'impact que peut avoir dans l'évaluation de requêtes, la connaissance de tels types.

Il sera aussi intéressant d'ajouter à PBE différentes fonctionnalités comme l'aide à l'écriture de DTD, ou de types ℂDuce, de considérer graphiquement les unions de types, d'avoir une représentation plus visuelle des tableaux de manière à ne pas devoir utiliser de variables, etc. Et dans un souci de standardisation, de proposer une réécriture des tableaux PBE en des requêtes XQuery, travail qui est en cours.

Table des figures

1.1 Exemple de document XML 12
1.2 Représentation sous forme d'arbre d'un document XML 12

2.1 Document XML de référence représentant une bibliographie 19
2.2 Représentation sous forme d'arbre du document XML de référence 20
2.3 DTD associée à un document XML de référence 21
2.4 Une spécification XML SCHEMA 23
2.5 Une page XHTML et sa représentation dans un navigateur 24
2.6 Représentation des déconstructeurs sur un arbre XML 26
2.7 Tableau récapitulatif des itérateurs et déconstructeurs de différents langages de manipulation de documents XML 29
2.8 Algèbre de motifs de SGMLQL 31
2.9 Requête Q1 des « *XML Query Use Cases* » 33
2.10 Algèbre de motifs et de types de XDUCE 35
2.11 DTD et types ℂDuce types représentant une bibliographie 39
2.12 Une valeur ℂDuce représentant le document XML de référence . . 40
2.13 Quelques cas d'inférence de types par ℂDuce. 44
2.14 Algèbres des types et motifs de ℂDuce 45
2.15 Représentation d'utilisation conjointe des deux déconstructeurs sur un arbre XML 45

3.1 Règle de typage de l'opérateur `transform` 54
3.2 Règle de typage pour les requêtes ℂQL 55
3.3 Règle de typage pour la projection sur descendants 56
3.4 Requête Q5 des « *XML Query Use Cases* »et le type de la valeur `bstore2` 57
3.5 Requête Q5 des "*XML Query Use Cases*" utilisant le filtrage par motifs 58
3.6 Quels sont les livres n'ayant pas un seul auteur ? 60
3.7 Requête Q1 utilisant le filtrage par motifs 61

3.8 Requête Q1 utilisant la navigation par projection 61
3.9 Relation entre les différents éléments d'un document XMark sous forme d'arbre 62
3.10 Requête Q1 utilisant la navigation par projection et le filtrage par motifs 63
3.11 DTD avec entités récursives 63

4.1 Requête avec projections 66
4.2 Requête avec un niveau d'imbrication 67
4.3 Grammaire des requêtes avec et sans projections 67
4.4 Contextes d'évaluation 68
4.5 Définition du langage cible 71
4.6 Définition de la réduction $\rightarrow$ 71
4.7 Définition du contexte $R[]$ 71
4.8 Requête après la traduction $\mathcal{T}$ 83
4.9 Application de Θ sur la requête après la traduction $\mathcal{T}$ 85
4.10 Consolidation des motifs 86
4.11 Requête avec condition disjonctive 87
4.12 Requête avec condition remontée 88

5.1 Résumé des résultats des tests sur les requêtes des « *XML Query Use Cases* » 92
5.2 Temps d'exécution (sans temps de chargement) pour la requête Q1 94
5.3 Requête Q1 - Donner les titres et années de publication des livres publiés par "Addison-Wesley" après 1991. 95
5.4 Requête Q2 - Donner chaque paire titre/auteur pour chaque livre de la bibliographie. 96
5.5 Requête Q3 - Créer pour chaque livre de la bibliographie un nouvel élément `<result>` contenant le titre et les auteurs. 97
5.6 Temps d'exécution pour la requête Q2 98
5.7 Temps d'exécution pour la requête for Q3 98
5.8 Requête Q4 - Donner pour chaque auteur la liste des titres des livres auxquels ils ont participés, groupés dans une balises `<result>` . . . 99
5.9 Temps d'exécution (sans temps de chargement) pour la requête Q4 100
5.10 Requête Q5 - Faire un comparatif des prix pour un même titre des livres de la bibliographie et de bstore2. 101
5.11 Temps d'exécution pour la requête Q5 [sans QEXO] 102
5.12 Différentes expérimentations sur les jointures 103

5.13 Résumé des résultats des tests XMark 104
5.14 Requête XMark Q1 - Renvoyer les noms des items avec l'id 'item20748' enregistrés en Amérique du Nord. 105
5.15 Requête XMark Q8 - Donner la liste des noms des personnes et le nombre d'items qu'ils ont achetés. 106
5.16 Requête XMark Q12 - Pour chaque personne, donner la liste du nombre d'items qu'ils ont en vente dont le prix ne dépasse pas 0.02% de leur revenu. 108
5.17 Requête XMark Q16 - Renvoyer les id des vendeurs d'enchères fermées qui ont au moins un mot-clefs en emphase. 109
5.18 Squelettes des documents **exponential2.xml**, **layered.xml** et **mixed.xml** utilisés dans les mesures. 110
5.19 Temps complets pour la partie XPath sur le document *exponential2.xml*. 116
5.20 Différents temps pour la requête **Q1.1**(n) **sur différents documents.** 117
5.21 Temps complets pour la partie XPath sur le document *exponential2.xml* en présence d'une DTD précise 118
5.22 Temps complets pour la partie XQuery sur le document *mixed.xml.* 119
5.23 Tableau comparatif des résultats de la partie XPath sur le document *exponential2.xml*. 120

6.1 Une requête QBE : Donner tous les employés dont le salaire est compris entre 10.000 et 15.000, et qui travaillent dans le département qui vend des stylos. 126
6.2 Requêtes XQBE Q1 (a), Q2 (b) et Q3 (c) 127
6.3 DTD de référence pour PBE 130
6.4 Document XML de référence pour PBE 131
6.5 Création d'un tableau de filtrage 131
6.6 Tableau de filtrage créé portant sur la variable `doc` 132
6.7 Ajout de la variable `books` dans le tableau de filtrage. 133
6.8 Création du tableau de construction q1 133
6.9 La requête q1 déduite en $\mathbb{C}$QL. 134
6.10 Retourner tous les livres 134
6.11 Retourner tous les titres de livres 135
6.12 Requête $\mathbb{C}$QL et la réponse. 135
6.13 Exemple de requête imbriquée 136
6.14 Les requêtes q3 et q4 déduites en $\mathbb{C}$QL. 138

6.15 Exemple de tableaux à plusieurs lignes 138

6.16 Résultat de la requête 139

6.17 Définition des types ℂDuce utilisés pour les en-tête de tableaux de filtrage 139

6.18 Règles d'inférence pour les requêtes PBE non imbriquées et sans boîte de condition. 145

6.19 la fonction pattern() 146

6.20 Mettre le titre et le prix de chaque livre dans une élément `<result>` 146

6.21 Règles d'inférence pour les requêtes PBE non imbriquées et sans boîte de condition avec le retrait de la redondance. 148

6.22 Règles d'inférence pour les requêtes imbriquées sans condition . . . 149

6.23 Mettre dans un élément `<result>` les titres et les auteurs, en changeant la balise `author` par la balise `auteur`. 150

6.24 Règles d'inférence pour les requêtes imbriquées sans condition avec retrait de la redondance. 151

6.25 Exemple de requête imbriquée telle que l'ensemble des variables à déclarer de l'une soit le même que dans la seconde. 152

6.26 Règles d'inférence pour les requêtes imbriquées avec condition (partie I). 154

6.27 Règles d'inférence pour les requêtes imbriquées avec condition (partie II). 155

6.28 Règles d'inférence pour les requêtes imbriquées avec condition (partie III). 156

6.29 Quels sont les titres qui apparaissent dans la bibliographie et dans `bstore2`? 157

6.30 Quels sont les titres des livres dont le prix est supérieur à 50. . . . 158

6.31 Règles d'inférence pour les requêtes imbriquées avec condition et recherche descendante. 159

Bibliographie

[ABM05] Andrei Arion, Véronique Benzaken, and Ioana Manolescu. XML Access Modules : Towards Physical Data Independence in XML Databases. In *XIME-P*, 2005. (cité page 162)

[ABMV05] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Ravi Vijay. ULoad : Choosing the Right Storage for Your XML Application. In *VLDB*, pages 1330–1333, 2005. (cité page 162)

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : from Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000. (cité page 11)

[AFMZ06] L. Afanasiev, M. Franceschet, M. Marx, and E. Zimuel. Xcheck : A platform for benchmarking XQuery engines (demo), 2006. (cité page 110)

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. (cité page 126)

[AM05] Loredana Afanasiev and Maarten Marx. XCheck, an automated XQuery benchmark tool, 2005. `http://ilps.science.uva.nl/Resources/XCheck`. (cité page 110)

[AMM05a] Loredana Afanasiev, Ioana Manolescu, and Philippe Michiels. Member : A micro-benchmark repository, 2005. `http://ilps.science.uva.nl/Resources/MemBeR/`. (cité page 110)

[AMM05b] Loredana Afanasiev, Ioana Manolescu, and Philippe Michiels. Member : A micro-benchmark repository for XQuery. In *XSym*, pages 144–161, 2005. (cité page 110)

[AMMM06] Loredana Afanasiev, Ioana Manolescu, Cédric Miachon, and Philippe Michiels. Micro-benchmarking XQuery experiments page, 2006. `http://www-rocq.inria.fr/~manolesc/microbenchmarks.html`. (cité pages 111, 115)

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1) :68–88, 1997. (cité pages 13, 30)

[BCC04] Daniele Braga, Alessandro Campi, and Stefano Ceri. Xqbe : A graphical interface for xquery engines. In *EDBT 2004*, volume 2992 of *Lecture Notes in Computer Science*, pages 848–850, 2004. (cité page 126)

[BCC05] Daniele Braga, Alessandro Campi, and Stefano Ceri. "XQBE (XQuery By Example) : A visual interface to the standard XML query language". In *Transactions on Database Systems*, volume 30, pages 398–443, 2005. (cité page 126)

[BCCN06] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyễn. Type-Based XML Projection. In *32nd International Conference on Very Large Databases*, 2006. (cité pages 104, 121, 163)

[BCD+99] Luc Bouganim, Tatiana Chan-Sine-Ying, Tuyet-Tram Dang-Ngoc, Jean-Luc Darroux, Georges Gardarin, and Fei Sha. Miro web : Integrating multiple data sources through semistructured data types. In *The VLDB Journal*, pages 750–753, 1999. (cité page 126)

[BCF+01] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0 : An XML Query Language, 2001. (cité pages 13, 32, 91)

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce : an XML-friendly general purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press. (cité pages 13, 36, 38, 47, 55)

[BCL+05] Stéphane Bressan, Barbara Catania, Zoé Lacroix, Ying Guang Li, and Anna Maddalena. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2) :211–240, 2005. (cité page 104)

[BCM04] Véronique Benzaken, Giuseppe Castagna, and Cédric Miachon. CQL : a pattern-based query language for XML. In *Actes des 20ès Journées des Bases de Données Avancées (BDA)*, 2004. (cité pages 48, 57, 90)

[BCM05] Véronique Benzaken, Giuseppe Castagna, and Cédric Miachon. A full pattern-based paradigm for XML query processing. In *PADL 05, 7th International Symposium on Practical Aspects of Declarative Languages*, number 3350 in Lecture Notes in Computer Science, pages 235–252. Springer, January 2005. (cité pages 48, 57, 90)

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an object-oriented database system : the story of O_2*. Morgan Kaufmann, 1992. (cité page 29)

[BDL+01] S. Bressan, G. Dobbie, Z. Lacroix, M. Lee, Y. Li, U. Nambiar, and B. Wadhwa. X007 : Applying 007 benchmark to XML query processing tool. In *CIKM*, pages 167–174, 2001. (cité page 110)

[Bel] Bell-labs. *Galax*. `http://db.bell-labs.com/galax/`. (cité pages 91, 111)

[BH05] Don Box and Anders Hejlsberg. *The LINQ Project Overview, .NET Language Integrated Query*, September 2005. (cité page 34)

[BMS05] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in cω. In *ECOOP 2005*, volume 3586 of *Lecture Notes in Computer Science*, pages 287 – 311, Aug 2005. (cité page 34)

[Bot] P. Bothner. Qexo - the GNU Kawa implementation of XQuery. http ://www.gnu.org/software/qexo/. (cité page 91)

[Bot04] P. Bothner. Compiling XQuery to java bytecodes. In I. Manolescu and Y. Papakonstantinou, editors, *Proceedings of the First Int. Workshop on XQuery Implementation, Experience and Perspectives* <XIME-P/>, pages 31–37, 2004. (cité page 91)

[BR01] Timo Böhme and Erhard Rahm. Xmach-1 : A benchmark for XML data management. In *Proceedings of BTW2001, Oldenburg, 7.-9. März, Springer, Berlin*, March 2001. (cité page 110)

[Cam92] Mary V Campbell. *Microsoft Access - Inside and Out*. Osborne McGraw-Hill, 1992. (cité page 125)

[Cas05] Giuseppe Castagna. Patterns and types for querying XML documents. In *DBPL 2005*, volume 3774 of *Lecture Notes in Computer Science*, pages 1–26, Dec 2005. (cité pages 13, 14, 25)

[CD99] J. Clark and S. DeRose. *XML Path Language (XPath)*. W3C Recommendation, `http://www.w3.org/TR/xpath/`, November 1999. (cité pages 13, 26)

[CDF01] Sara Comai, Ernesto Damiani, and Piero Fraternali. Computing graphical queries over XML data. *ACM Trans. Inf. Syst.*, 19(4) :371–430, 2001. (cité page 127)

[CDu05] The CDucers. ℂDuce : an XML-centric programming language. In *Actes des 21es Journées des Bases de Données Avancées (BDA)*, 2005. (cité page 57)

[CF05] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, Lisboa, Portugal, 2005. ACM Press. Joint ICALP-PPDP keynote talk. (cité page 70)

[CFF+03] Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases. Technical Report 20030822, World Wide Web Consortium, 2003. (cité pages 18, 33, 57, 60, 90, 127, 155)

[CFMR03] Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. XML query (XQuery) requirements. Technical Report 20030627, World Wide Web Consortium, 2003. (cité page 32)

[CGA+02] G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. "The Query Language TQL". In *In 5th Int. Workshop on the Web and Databases (WebDB)*, 2002. (cité page 29)

[CHMW96] Michael J. Carey, Laura M. Haas, Vivekananda Maganty, and John H. Williams. Pesto : An integrated query/browser for object databases. In *VLDB*, pages 203–214, 1996. (cité page 126)

[CKK+99] Sara Cohen, Yaron Kanza, Yakov A. Kogan, Werner Nutt, Yehoshua Sagiv, and Alexander Serebrenik. Equix easy querying in XML databases. In *WebDB (Informal Proceedings)*, pages 43–48, 1999. (cité page 126)

[Cla99] James Clark. *XSL Transformations (XSLT).* W3C Recommendation, `http://www.w3.org/TR/xslt/`, November 1999. (cité pages 12, 20)

[Cla01] James Clark. *TREX : Tree Regular Expressions for XML*, 2001. www.thaiopensource.com/trex/. (cité page 20)

[Clu98] Sophie Cluet. Designing OQL : allowing objects to be queried. *Inf. Syst.*, 23(5) :279–305, 1998. (cité page 29)

[CRF01] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt : An XML query language for heterogeneous data sources. In *WebDB 2000 (selected papers)*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25, 2001. (cité page 33)

[csh05] *C# Version 3.0 Specification*, September 2005. (cité page 34)

[DFF+98] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "XML-QL : A Query Language for XML". In *WWW The Query Language Workshop (QL)*, Cambridge, MA, , 1998. (cité pages 14, 31, 33, 126)

[DFF+99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. *Computer Networks (Amsterdam, Netherlands : 1999)*, 31(11–16) :1155–1169, 1999. (cité pages 14, 31)

[DFM+02] D. Draper, P. Frankhauser, M.Fernandez, A. Malhotra, K. Rose, M.Rys, J.Simeon, and P.Wadler. XQuery 1.0 Formal Semantics. Technical report, World Wide Web Consortium, Mar 2002. (cité page 33)

[Erw03] M. Erwig. Xing : A visual xml query language, 2003. (cité page 126)

[exi] *eXist : Open Source Native XML Database.* `http://exist.sourceforge.net/`. (cité page 111)

[exp] *The Expat XML Parser.* http ://expat.sourceforge.net/. (cité page 104)

[FCB02] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002. (cité pages 36, 54, 59, 173)

[FCB05] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. Extended version of [FCB02], in preparation., 2005. (cité page 38)

[FLdS01] Irna M. R. Evangelista Filha, Alberto H. F. Laender, and Altigran S. da Silva. Querying Semistructured Data By Example : The Qsbye Interface. In *Workshop on Information Integration on the Web*, pages 156–163, 2001. (cité page 126)

[Fra] X. Franc. Qizx/open. http ://www.xfra.net/qizxopen. (cité pages 91, 111)

[Fri04a] A. Frisch. Regular tree language recognition with static information. In *Proc. Int Workshop on Programming Languages for XML (PlanX 04)*, 2004. (cité pages 36, 53)

[Fri04b] A. Frisch. Regular tree language recognition with static information. In *Proc. IFIP Conference on Theoretical Computer Science (TCS)*, Toulouse, 2004. Kluwer. (cité page 53)

[Fri04c] Alain Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML.* PhD thesis, Université Paris 7, December 2004. (cité pages 36, 55, 80)

[FSW+99] M. Fernández, J. Siméon, P. Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, and J. Widom. XML query languages : Experiences and exemplars. 1999. (cité page 91)

[FSW00] Mary Fernández, Jérôme Siméon, and Philip Wadler. An algebra for XML query. In *Foundations of Software Technology and Theoretical Computer Science*, number 1974 in Lecture Notes in Computer Science, pages 11–45, 2000. (cité page 53)

[GLPS05] Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. XML goes native : Run-time representations for Xtatic. In *14th International Conference on Compiler Construction*, April 2005. (cité pages 13, 36)

[Hos01] H. Hosoya. *Regular Expression Types for XML.* PhD thesis, The University of Tokyo, 2001. (cité page 34)

[HP00] H. Hosoya and B.C. Pierce. XDuce : A typed XML processing language. In *WebDB2000, 3rd International Workshop on the Web and Databases*, 2000. (cité page 13)

[HP01] H. Hosoya and B.C. Pierce. Regular expression pattern matching for XML. In *POPL '01, 25th ACM Symposium on Principles of Programming Languages*, 2001. (cité pages 34, 35)

[HP03] H. Hosoya and B. Pierce. XDuce : A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2) :117–148, 2003. (cité page 34)

[HVP00] H. Hosoya, J. Vouillon, and B. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000. (cité page 34)

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1) :46–90, 2005. (cité page 33)

[ISO86] ISO. *8879 : Information Processing - Text and Office Systems - Standardized Markup Language(SGML)*, October 1986. International Organization for Standardization. (cité pages 11, 18)

[KMS02] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. The DSD schema language. *Automated Software Engineering*, 9(3) :285–319, 2002. Kluwer. Earlier version in Proc. 3rd ACM SIGPLAN-SIGSOFT Workshop on Formal Methods in Software Practice, FMSP '00. (cité page 20)

[LC00] Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(3) :76–87, 2000. (cité page 20)

[LMR95] Jacques Le Maitre, Elisabeth Murisasco, and Monique Rolbert. SgmlQL, un langage d'interrogation de documents SGML. In *Actes des 11es Journées des Bases de Données Avancées (BDA)*, 1995. (cité pages 29, 30)

[MB05] Erik Meijer and Brian Beckman. XLinq : XML Programming Refactored (The Return of The Monoids). In *XML*, 2005. (cité page 34)

[MLM01] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001. (cité page 20)

[MMM06] Ioana Manolescu, Cédric Miachon, and Philippe Michiels. Towards microbenchmarking XQuery. In *First International Workshop on Performance and Evaluation of Data Management Systems (EXPDB 2006)*, Chicago, 2006. (cité pages 110, 119, 122)

[mon] *MonetDB : Query Processing at Light-Speed.* `http://monetdb.cwi.nl/`. (cité page 111)

[MP00] Kevin D. Munroe and Yannis Papakonstantinou. BBQ : A visual interface for integrated browsing and querying of XML. In *VDB*, 2000. (cité page 126)

[MS03] Amélie Marian and Jérôme Siméon. Projecting XML Documents. In *VLDB*, pages 213–224, 2003. (cité pages 104, 163)

[Mur00] Makoto Murata. *RELAX (Relax Language Description for XML)*, 2000. http ://www.xml.gr.jp/relax. (cité page 20)

[Ngu04] Kim Nguyễn. Une algèbre de filtrage pour le langage ℂDuce. DEA *Programmation*, Université Paris-Sud 11, September 2004. Available at `http://www.lri.fr/~kn/main.pdf`. (cité pages 57, 122, 163)

[oca] *Objective Caml.* http ://caml.inria.fr/ocaml/. (cité page 91)

[PPV05] Michalis Petropoulos, Yannis Papakonstantinou, and Vasilis Vassalos. Graphical query interfaces for semistructured data : the QURSED system. *j-TOIT*, 5(2) :390–438, May 2005. (cité page 126)

[PV00] Yannis Papakonstantinou and Victor Vianu. DTD Inference for Views of XML Data. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 35–46, Dallas, Texas, 2000. (cité page 29)

[pxp] *PXP : the XML parser for OCaml.* http ://www.ocaml-programming.de/programming/pxp.html. (cité page 104)

[rel] RELAX NG specification. http ://www.oasis-open.org/committees/relax-ng/spec-20011203.html. (cité pages 20, 22)

[RG02] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. WCB/McGraw-Hill, 3rd edition, 2002. (cité pages 100, 126)

[RLS98] A. J. Robie, J. Lapp, and D. Schach. "XML Query Language (XQL). In *WWW The Query Language Workshop (QL)*, Cambridge, MA, , 1998. (cité page 33)

[sax] *SAXON : The XSLT and XQuery Processor.* `http://saxon.sourceforge.net/`. (cité page 111)

[SWK+01] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001. (cité pages 89, 103)

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark : A Benchmark for XML Data Management. In *Proceedings of the Int'l. Conference on Very Large Database Management (VLDB)*, pages 974–985, 2002. (cité pages 62, 89)

[W3C99] W3C Recommendation. *HTML 4.01 Specification*, 1999. http ://www.w3.org/TR/html4. (cité page 23)

[W3C00] W3C Recommendation. *Extensible Markup Language (XML) 1.0*, (second edition) edition, 2000. (cité pages 11, 18, 20, 21)

[W3C01a] W3C Recommendation. *Mathematical Markup Language (MathML) Version 2.0*, 2001. http ://www.w3.org/TR/MathML2/. (cité page 24)

[W3C01b] W3C Recommendation. *XML Schema*, 2001. http ://www.w3.org/TR/xmlschema-0/. (cité pages 20, 22)

[W3C02] W3C Recommendation. *XHTML 1.0 The Extensible HyperText Markup Language*, second edition, 2002. http ://www.w3.org/TR/xhtml1. (cité page 23)

[W3C04] W3C Recommendation. *Scalable Vector Graphics (SVG) 1.1*, 2004. http ://www.w3.org/TR/SVG. (cité page 24)

[xli] *XLinq Overview.* http ://msdn.microsoft.com/VBasic/Future/XLinq (cité page 34)

[YÖK04] B. Yao, T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *ICDE*, pages 621–633, 2004. (cité page 110)

[Zlo75] Moshé M. Zloof. Query by example. In *AFIPS NCC*, pages 431–438, 1975. (cité pages 15, 125)

[Zlo77] Moshé M. Zloof. Query-by-example : A data base language. *IBM Systems Journal*, 16(4) :324–343, 1977. (cité pages 15, 125)