Académie de Montpellier

Université Montpellier II
— Sciences et Techniques du Languedoc —

Thèse

présentée à l'Université des Sciences et Techniques du Languedoc
pour obtenir le diplôme de DOCTORAT

Spécialité	: Informatique
Formation Doctorale	: *Informatique*
Ecole Doctorale	: *Information, Structures, Systèmes*

Un modèle de vues pour l'intégration de sources de données XML : VIMIX

par

Xavier Baril

Soutenue le 11 décembre 2003 devant le Jury composé de :

Danièle Hérin, Professeur, Université Montpellier II, ... Présidente du Jury

Zohra Bellahsène, Maître de conférences, HDR, Université Montpellier II, ... Directrice de Thèse

Claude Chrisment, Professeur, Université Toulouse III, ... Rapporteur
Christine Collet, Professeur, ENSIMAG, ... Rapporteur
Jacques Le Maitre, Professeur, Université de Toulon et du Var, ... Examinateur

à mes parents

Remerciements

Cette thèse a été préparée dans le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) dont je tiens à remercier l'ensemble des membres pour leur accueil et particulièrement Zohra Bellahsène et Danièle Hérin qui m'ont accueilli dans leur équipe.

Je tiens à exprimer ma profonde gratitude aux membres de mon jury pour l'honneur qu'ils m'ont fait en acceptant de juger mon travail.

Je remercie Claude Chrisment, Professeur à l'Université Paul Sabatier à Toulouse, d'avoir accepté d'être rapporteur de cette thèse. Ses remarques pertinentes et constructives sur mon travail m'ont aidé à en améliorer la qualité.

Je remercie Christine Collet, Professeur à l'ENSIMAG à Grenoble, d'avoir accepté d'être rapporteur de cette thèse. Les échanges que nous avons eus m'ont permis de prendre du recul et d'améliorer la qualité du mémoire. Je lui exprime ma sincère gratitude pour le temps et l'intérêt qu'elle m'a consacré.

Je remercie Jacques Le Maitre, Professeur à l'Université de Toulon et du Var, d'avoir accepté d'être examinateur de mon travail. Je lui exprime ma gratitude pour l'honneur qu'il m'a fait en participant à ce jury.

Je remercie Zohra Bellahsène, Maître de Conférences à l'Université de Montpellier II, qui a dirigé cette thèse. Je lui exprime ma sincère reconnaissance pour le temps et l'énergie qu'elle m'a consacrés. Sa disponibilité et son soutien m'ont été précieux pendant ces années.

Je remercie Danièle Hérin, Professeur à l'Université Montpellier II, d'avoir accepté d'être présidente de ce jury. J'ai apprécié ses qualités humaines et la confiance qu'elle m'a témoignée pendant mon séjour au LIRMM. Qu'elle soit assurée ici de ma profonde reconnaissance.

Merci également aux (ex-)doctorants et aux membres permanents ou non du laboratoire avec qui j'ai partagé des discussions (scientifiques ou non), des repas, des cafés et des bons moments. Merci également à ceux avec que j'ai rencontré pendant mes enseignements à l'ENSCM (Ecole Nationale Supérieure de Chimie de Montpellier) et à l'ISIM (Institut des Sciences de l'Ingénieur de Montpellier). Il est difficile de citer tout le monde alors toutes mes excuses par avance à ceux que j'ai oublié. Je

remercie particulièrement Nicolas Vidot pour son amitié et son aide précieuse. Merci aussi à Didier, Lylia, Mathieu, Pierre-Alain, Fabien, Simon, Stéphane, Olivier, Stéphanie, Sébastien.

Je tiens à remercier également les membres de l'équipe SIG à l'IRIT où j'avais effectué mon stage de DEA. Merci à Olivier qui avait encadré mon travail pendant sa thèse en me donnant beaucoup de son temps.

Je tiens également à remercier ici tous les amis qui m'ont soutenu pendant ma thèse. Je pense plus particulièrement à Charly, Nicolas, Eric, Fabien, Gary, Marianne, Sylvain, Laurent, la famille Balmo et les copains de la Peña.

Enfin, je tiens à remercier très sincèrement mes parents. Leur soutien m'a été précieux pendant toutes mes études.

Sommaire

Chapitre 1

Introduction

1.1 Problématique de l'intégration de données

L'objectif d'un système d'intégration est de fournir une interface qui permet d'accéder de manière unifiée à différentes sources de données. Ces sources peuvent être hétérogènes et réparties sur un réseau informatique. Une vue unifiée de ces sources permet d'y accéder indépendamment de leur localisation et du format de leurs données. Considérons par exemple les différentes sources de données bibliographiques de plusieurs laboratoires scientifiques. L'intégration de ces sources permettrait d'avoir une vue d'ensemble de leurs publications.

Cependant, l'intégration de sources de données n'est pas une chose facile. En effet, les sources sont souvent hétérogènes car elles ont été définies indépendamment les unes des autres. Le processus d'intégration doit donc permettre de traiter des sources qui ont des modèles de données et/ou des schémas différents. Ensuite, il faut définir une architecture pour le système d'intégration.

Le processus d'intégration repose généralement sur un mécanisme de vues. Un tel mécanisme permet de restructurer des données en définissant différents points de vues de celles-ci. Il peut être utilisé pour personnaliser ou assurer la confidentialité des données dans un SGBD, mais également pour intégrer des données provenant de sources hétérogènes. Les modèles de vues reposent généralement sur les langages de requêtes utilisés pour interroger les données.

Récemment, le langage XML a été proposé pour échanger des données sur le *Web*. C'est un langage de balisage qui permet de représenter la structure de données irrégulières. De plus, ces données sont auto-décrites, c'est à dire que leur schéma (éventuellement irrégulier) est contenu dans les données. Rapidement, XML s'est imposé comme modèle commun pour intégrer des données. Pour cette raison, la définition d'un mécanisme de vues pour XML est importante. En effet, bien que de nombreux langages de requêtes pour XML aient été proposés [XQu03, QL'98, RLS98, DFF$^+$98, FS98, CCD$^+$98, CRF00] il n'y a eu à notre connaissance que peu de travaux sur les modèles de vues pour XML [Abi99]

Dans [Vel02], un modèle de vues pour XML dans un système à grande échelle (à l'échelle du *Web*) a

été proposé. Ce modèle de vues est basé sur un paradigme “chemin à chemin”, c'est à dire que chaque chemin du schéma de la vue est associé (par des *mappings*) à un ou plusieurs chemins dans la source de données. Contrairement aux langages de définition classiques, la spécification des vues n'est pas faite à l'aide d'un langage de requête. C'est le concepteur qui définit une DTD abstraite qui correspond au schéma de la vue et le système génère ensuite les *mappings*. [Vel02] c'est surtout intéressé au processus de réécriture de requêtes exprimées sur la DTD abstraite pour les traduire en requêtes sur les données XML concrètes effectivement stockées.

Pourtant, l'intérêt d'un modèle de vues pour XML est double. Tout d'abord, Il permet de restructurer et de faciliter l'accès à des documents XML, comme dans le contexte classique des SGBD. Ensuite, il permet également d'utiliser XML pour langage commun pour intégrer des données. L'utilisation d'XML présente de nombreux avantages qui seront présentés dans le chapitre suivant (§ 2.4.1, page 25).

Actuellement, de nombreux systèmes pour intégrer des données utilisent XML comme modèle commun [Gar02, LVPV99, NACP01, GMT02, DHW01a, BEA]. Dans ces systèmes, les vues sont définies en utilisant les langages de requêtes proposés pour XML ou des sous-ensemble de ces langages. Généralement, l'effort de recherche consiste à proposer des techniques de réécriture de requêtes pour interroger les données intégrées. Cependant, la présentation du modèle de vues est peu évoquée. Ces systèmes seront présentés dans le chapitre suivant (§ 2.4.2, page 26).

1.2 Contribution

La principale contribution de notre travail est un modèle de vues pour XML. Ce modèle est appelé VIMIX : VIew Model for Integration of XML sources. Il est composé d'un modèle de données et d'un langage de spécification de vues.

Notre modèle de données permet de représenter les données XML dans un graphe. Ce graphe prend en compte les liens de composition et de référence entre les éléments. Les liens de référence permettent de naviguer dans les données XML à travers les attributs de type `IDREF(S)`.

Le langage de spécification de vues permet de restructurer des données XML provenant de sources multiples et hétérogènes [BB00]. Il fonctionne par *pattern-matching*, c'est à dire que les données des sources sont instanciées à partir d'un motif à rechercher. Les données ainsi extraites peuvent être représentées par des tables relationnelles et restructurées avec des opérations d'union et de jointure. Enfin, le résultat d'une vue est spécifié par un arbre décrivant sa structure et ses données à l'aide d'expressions. Ces expressions permettent d'utiliser des fonctions d'agrégation pour créer de nouveaux éléments ou attributs dans le résultat de la vue.

Afin de faciliter la définition de motifs sur les sources, nous avons proposé un mécanisme d'aide [BB01a]. Ce mécanisme permet au concepteur de la vue de découvrir de manière partielle la structure de la source de données sur laquelle il est en train de définir un motif. Pour cela, nous pouvons utiliser deux sources d'information : la DTD de la source si elle existe, ou un résumé des données de la source appelé *dataguide*.

Les vues VIMIX permettent de définir le schéma médiateur d'un système d'intégration avec une approche mixte *GAV* et *LAV*. Ces deux approches sont présentées dans le chapitre 2 consacré à l'état de l'art du domaine des systèmes d'intégration.

Nous avons également proposé une méthode de matérialisation qui permet de stocker des vues VIMIX dans un SGBD relationnel. La définition de vues matérialisées permet de construire un entrepôt de données [BB03a]. Cette méthode sépare les données XML des sources qui sont stockées en utilisant un schéma générique et les *mappings* qui permettent de construire le résultat des vues. Cette méthode de matérialisation permet de maintenir les données de manière incrémentale. De plus, la méta modélisation utilisée pour le stockage permet de traiter de manière efficace les requêtes.

Enfin, nous avons implémenté notre approche. Le système DAWAX est un prototype qui permet de spécifier des vues VIMIX pour la construction d'un entrepôt de données XML. L'interface graphique proposée pour la spécification des vues implémente les mécanismes d'aide proposés pour fournir des assistants. Les requêtes SQL pour la construction et la maintenance de l'entrepôt dans le SGBD POSTGRESQL sont générées à partir de la spécification des vues VIMIX.

1.3 Organisation du mémoire

Dans le chapitre 2, nous présentons un état de l'art du domaine des systèmes d'intégration basés sur XML. Nous présentons le concept de vue qui est utilisé pour les systèmes d'intégration de données. Les deux approches principales (*GAV* et *LAV*) pour la définition du schéma médiateur sont également présentées. Enfin nous présentons un tour d'horizon des systèmes basés sur XML pour l'intégration de données.

Dans le chapitre 3, nous présentons notre modèle de vues VIMIX : (VIew Model for Integration of XML sources) . Notre modèle de données pour XML permet de représenter les données dans un graphe qui prend en compte les liens de composition et de référence des éléments. Pour spécifier les données à extraire, nous utilisons des motifs sur les sources. De plus, les données extraites peuvent être intégrées à l'aide d'opérations d'union et de jointure. Enfin, le résultat d'une vue est décrit à l'aide d'un arbre qui définit sa structure et ses données. Des nœuds de cet arbre permettent de construire la structure avec de nouveaux éléments et attributs. Les nœuds spécifiant les données du résultat contiennent des expressions permettant de manipuler les données extraites des sources.

Dans le chapitre 4, nous montrons comment VIMIX peut être utilisé pour définir le schéma médiateur d'un système d'intégration de données. Pour cela, nous utilisons une collection de vues VIMIX et les schémas de ces vues sont utilisés pour construire le schéma médiateur du système d'intégration. Enfin, nous proposons un mécanisme d'aide pour la spécification des motifs sur les sources.

Dans le chapitre 5, nous présentons une technique de stockage qui permet de matérialiser des vues VIMIX. Cette matérialisation permet de construire un entrepôt de données intégrant des sources XML dans un SGBD relationnel. Notre méthode de stockage sépare les données XML extraites des sources et les *mappings* qui permettent de construire le résultat des vues. Les données XML sont stockées dans un schéma générique qui permet d'utiliser un SGBD relationnel pour stocker des données XML. De plus, nous montrons que notre méthode de stockage facilite la maintenance de l'entrepôt et son interrogation.

Dans le chapitre 6, nous présentons le système que nous avons développé pour construire un entrepôt de données avec des vues VIMIX. Ce système est appelé DAWAX : DA*ta*WA*rehouse for* XML. Il propose une interface graphique pour spécifier des vues VIMIX. Cette interface permet d'utiliser des assistants qui sont basés sur les mécanismes d'aide proposés pour la spécification de motifs sur les sources. A partir de la spécification de vues VIMIX, DAWAX permet de construire un entrepôt de données en utilisant le SGBD POSTGRESQL et de maintenir les données à jour.

Enfin, le chapitre 7 contient la conclusion. Il présente un bilan de notre travail. Des perspectives qui permettraient de compléter notre approche sont aussi évoquées.

Chapitre 2

Vues et intégration de données XML

2.1 Introduction

Dans ce chapitre, nous présentons un état de l'art du domaine des systèmes d'intégration de données basés sur XML. Les systèmes d'intégration de données sont généralement basés sur un **mécanisme de vues** que nous présentons ici. Un mécanisme de vues permet de définir un schéma médiateur qui présente une vue unifiée de sources de données hétérogènes.

Il existe principalement deux approches pour construire le **schéma médiateur** d'un système d'intégration. L'approche *GAV* définit le schéma médiateur comme une collection de vues sur les sources. A l'inverse, avec l'approche *LAV* le schéma médiateur est construit indépendamment des sources. Les sources sont ensuite définies comme des vues sur le schéma médiateur.

La définition d'un schéma médiateur sur des sources hétérogènes nécessite l'utilisation d'un **modèle commun**. Le langage XML présente les qualités nécessaires pour être utilisé comme modèle commun pour intégrer des données provenant de sources hétérogènes. En effet, XML permet de représenter des données semistructurées. Les modèles de données semistructurées permettent de représenter des données dont la structure est irrégulière.

Dans notre thèse, nous supposons que les sources de données sont capables d'exporter des données XML. Ce choix semble raisonnable dans le contexte actuel [Gar02, MBV03, DHW01a]. Pour permettre d'intégrer des sources XML, nous avons proposé un modèle de vues pour XML : VIMIX. De plus, un modèle de vues pour XML peut être utile dans d'autres applications, par exemple pour personnaliser des données ou gérer des contraintes de confidentialité.

Ce chapitre est organisé comme suit.

- Dans la section 2.2, nous présentons le concept de vues. Une vue est généralement définie comme un triplet (domaine, schéma, définition). Nous présentons également un tour d'horizon des problématiques étudiées dans le domaine des vues relationnelles, définies avec le langage SQL.

– Dans la section 2.3, nous présentons l'intérêt de définir un schéma médiateur pour intégrer des données provenant de sources hétérogènes. Nous présentons également les deux approches *GAV* et *LAV* qui permettent d'intégrer des données.
– Dans la section 2.4, nous présentons l'apport d'XML pour les systèmes d'intégration de données. Un tour d'horizon de différents systèmes basés sur XML pour intégrer des sources hétérogènes est également présenté.

2.2 La notion de vue

Nous présentons ici le concept de vues dans le contexte des bases de données. L'objectif fonctionnel d'un mécanisme de vues est de fournir différentes représentations d'une base de données, également appelées points de vues. Tout d'abord, nous montrons comment se positionne un mécanisme de vues dans l'architecture fonctionnelle d'une base de données. La Figure 2.1 présente les différents niveaux de l'architecture fonctionnelle d'une base de données :

– le niveau physique,
– le niveau logique,
– le niveau externe.

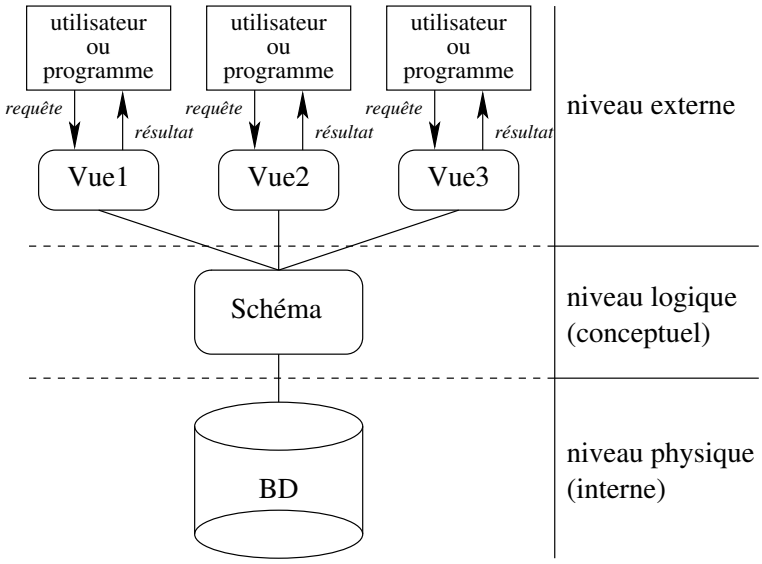

FIG. 2.1 – Architecture fonctionnelle d'une base de données.

Le niveau le plus bas est appelé **niveau physique** ou **interne**. Il gère le stockage des données de la base dans la mémoire secondaire de l'ordinateur. Pour cela il utilise le système de fichier et des mécanismes d'index.

Le **niveau logique** est aussi appelé parfois niveau conceptuel. Il décrit l'organisation logique des données, sans se préoccuper de leur stockage physique. Les systèmes de gestion de bases de données offrent une interface qui permet de manipuler les données stockées au niveau logique. Par exemple, pour les SGBD relationnels le langage SQL permet de définir le schéma et de manipuler les données de la base.

Enfin, le **niveau externe** permet de spécifier l'interface de la base de données avec les applications ou les utilisateurs. Un mécanisme de vues restructure les données stockées dans le SGBD, pour fournir différents **points de vues** des ces données à des utilisateurs ou à des programmes d'application.

Un mécanisme de vues permet donc de **restructurer** les données dans un SGBD. Cette restructuration peut être utilisée dans plusieurs buts :

- la personnalisation des données,
- assurer la confidentialité des données,
- faciliter l'utilisation de la base de données par des programmes.

Un mécanisme de vues peut être utilisé pour **personnaliser** les données. La restructuration permet de définir différents points de vues, correspondant chacun à des groupes d'utilisateurs. Ensuite, plutôt que de présenter aux utilisateurs le schéma de la base, on leur présente les vues qui ont été définies pour leur groupe. Ainsi, les utilisateurs ne voient que les informations qui leurs sont destinées, avec la structure qui a été définie pour eux.

De la même façon, un mécanisme de vues permet d'assurer la **confidentialité** d'une base de données. Le filtrage des données permet de fournir à un groupe d'utilisateurs seulement les données auxquelles on souhaite qu'ils aient accès.

Enfin, un mécanisme de vues peut être utilisé pour **faciliter l'accès aux données** de la base à des programmes d'application. La restructuration des données permet de faciliter le travail des programmeurs d'applications par exemple en calculant des jointures ou en utilisant les fonctions d'agrégation du langage de requête pour créer des attributs calculés.

De plus, un mécanisme de vues peut être utilisé pour **intégrer des données**. Nous montrerons dans la suite comment un tel mécanisme permet de définir un schéma médiateur qui présente une vue unifiée de sources hétérogènes.

Définition 2.1 (Vue) *Une vue est un triplet (domaine, schéma, définition) qui permet de restructurer des données et de fournir un schéma unifié de ces données.*

La Figure 2.2 illustre cette définition en présentant les éléments du concept de vue.

- Le **domaine** est l'ensemble des sources contenant les données de la vue. Sur la figure, les trois sources qui constituent le domaine de la vue sont encadrées.
- Le **schéma** est la structure utilisée par la vue pour présenter les données. Ce schéma joue le rôle d'interface entre les utilisateurs de la vue et les données des sources. Dans le contexte relationnel, le schéma de la vue est celui de la requête SQL qui permet de calculer le résultat.

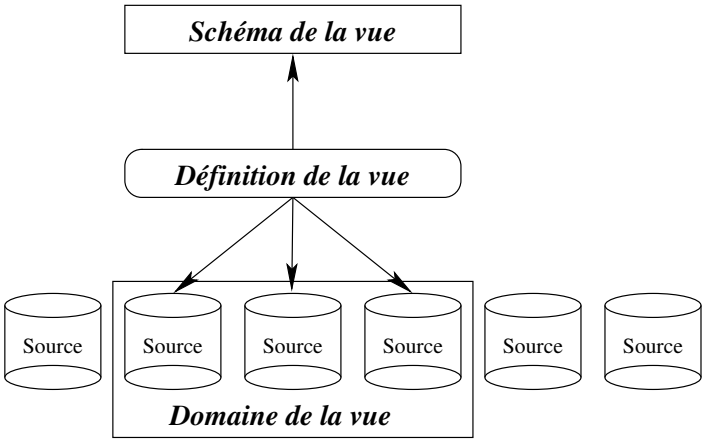

FIG. 2.2 – Définition d'une vue.

– La **définition** spécifie les données des sources qui seront associées au schéma de la vue. Dans le contexte relationnel, la définition de la vue est une requête SQL.

Il existe principalement deux stratégies pour gérer des vues dans un SGBD.

1. Stocker seulement la définition de la vue, alors on parle de vue virtuelle.
2. Calculer et stocker le résultat de la vue, on parle alors de vue matérialisée.

Lorsqu'une vue est **matérialisée**, son résultat est calculé puis stocké. L'avantage des vues matérialisées, c'est qu'elles permettent d'accéder rapidement aux données du résultat de la vue. Ces données peuvent être utilisées pour répondre à des requêtes. Cette technique nécessite une phase de réécriture, mais accélère le traitement des requêtes.

Cependant, l'inconvénient des vues matérialisées c'est qu'il faut maintenir à jour les données de la vue. En effet, lorsque les données des sources sont modifiées, les données de la vue doivent être modifiées pour rester cohérentes avec la définition de la vue. Pour cela, on peut utiliser des techniques de maintenance incrémentale qui permettent de propager les modifications et de recalculer seulement les données du résultat qui sont affectées par la modification.

Lorsqu'une vue est **virtuelle**, son résultat n'est pas stocké. Il doit donc être recalculé pour chaque accès aux données du résultat de la vue. L'avantage de cette technique, c'est qu'elle évite les problèmes de maintenance des données. Cependant, la phase de calcul des données peut être coûteuse lorsqu'on veut accéder aux données du résultat de la vue.

2.2.1 Vues dans le modèle relationnel

Les concepts mathématiques de l'**algèbre relationnelle** ont été définis pour le modèle relationnel. Nous allons rappeler ici quelques définitions de cette algèbre. Pour plus de détails, on pourra se reporter à des ouvrages de référence sur les systèmes de base de données [AHV95, UW97, GMUW02].

Définition 2.2 (Sorte d'une relation) *La sorte U d'une relation R est un ensemble non vide et fini d'attributs. On note sorte$(R) = U_R = \{A_1, \ldots, A_k\}$.*

Définition 2.3 (Schéma d'une relation) *Un schéma relationnel est un nom de relation R. Si $U_R = \{A_1, \ldots, A_k\}$, alors on note le schéma de cette relation $R(A_1, \ldots, A_k)$.*

Le schéma d'une base de données relationnelle est défini à partir des schémas de ses relations.

Définition 2.4 (Schéma d'une base de données) *Un schéma de base de données est un ensemble non vide et fini de schémas relationnels $B = \{R_1(U_{R_1}), \ldots R_n(U_{R_n})\}$*

Exemple Considérons la base de données `biblio` dont le schéma est présenté par la Figure 2.3. Le schéma de cette base de données est composé de trois relations. La schéma de la relation `Auteur`

```
Biblio = { Auteurs (id, nom, prenom, pays),
           Livres  (isbn, titre, editeur, annee),
           Ecrit   (id, isbn) }
```

FIG. 2.3 – Schéma relationnel d'une base de données bibliographique.

est composée d'attributs qui permettent de stocker les informations d'un auteur. L'attribut `id` permet d'identifier un auteur, les attributs `nom` et `prenom` son nom et son prénom, l'attribut `pays` sa nationalité. La relation `Livres` permet de représenter des livres. Enfin, la relation `Ecrit` permet d'associer les livres et leurs auteurs.

Soit la fonction *dom* qui définit le domaine d'un attribut, c'est à dire l'ensemble des valeurs qu'il peut prendre.

Définition 2.5 (Instance d'une relation) *Une instance de relation $I(R)$ est un sous-ensemble du produit cartésien du domaine des attributs de la relation : $I(R) \subseteq \prod_{A \in U_R} dom(A)$. Les éléments de l'instance d'une relation sont appelés tuples.*

On peut formuler des requêtes pour interroger une base de données grâce aux **opérateurs relationnels**. L'opération de sélection (notée σ) permet de "filtrer" les tuples d'une relation selon certains critères. L'opération de projection (notée π) permet de supprimer certains attributs des tuples. L'opération de jointure naturelle (notée $\bowtie$), permet de "croiser" les tuples provenant de plusieurs relations. L'opération de renommage (notée $\rightarrow$), permet de renommer les attributs de la sorte d'une relation. L'opération d'union (notée $\cup$), permet d'ajouter les tuples de deux relations de même sorte. L'opération de différence ensembliste (notée $-$), permet de supprimer les tuples d'une relation dans une autre relation de même sorte. Le résultat de chacune de ces opérations est une nouvelle relation. Nous ne détaillerons pas ici leur fonctionnement, pour cela on pourra se référer aux ouvrages de référence [AHV95, UW97, GMUW02].

Le langage SQL, basé sur l'algèbre relationnelle, permet d'écrire des requêtes de création, d'interrogation et de mise à jour d'une base de données relationnelle. Lorsque le terme requête est utilisé seul, il désigne généralement une requête d'interrogation. En SQL, comme dans les SGBD relationnels, le concept de relation est appelé table : chaque attribut est une colonne et chaque tuple une ligne de la table. Le résultat d'une requête SQL est une table dont la sorte est définie par la spécification de la requête.

Le langage SQL permet de définir des vues sur une base de données. Pour cela, il propose une clause `CREATE VIEW` qui permet de définir une vue comme le étant le résultat d'une requête. Le schéma de la vue sera défini par son nom et la sorte du résultat de la requête.

Exemple Considérons la base de données `biblio` définie précédemment. On désire créer une vue des auteurs français sur cette base de données. Cette vue peut être définie par le code SQL suivant :

```
CREATE VIEW Auteurs_F AS
SELECT nom, prenom
FROM   Auteurs
WHERE  pays = "France"
```

La clause `CREATE VIEW` permet de définir le nom de la vue : `Auteurs_F`. Ensuite, la requête SQL permet de spécifier la sélection des auteurs dont la nationalité est française. La clause `SELECT` définit la sorte du résultat de la vue qui sera composé par le nom et le prénom. La clause `FROM` indique que le domaine de la vue est la table `Auteurs`. Enfin, la clause `WHERE` contient une condition qui permet de sélectionner seulement les auteurs dont le pays est la France. Le schéma de cette vue sera :
`Auteurs_F(nom, prenom)`

Nous allons maintenant présenter un tour d'horizon des différentes problématiques qui ont été étudiées dans le contexte des vues relationnelles.

Réécriture de requêtes en utilisant les vues

L'utilisation de vues pour répondre à des requête est une technique utile pour résoudre de nombreux problèmes de bases de données : l'optimisation de requêtes, la gestion de l'indépendance physique des données, l'intégration de données et la conception d'entrepôts de données [Hal01, Hal00]. L'utilisation de vues pour répondre à des requêtes utilise la technique de **réécriture de requêtes**. Pour cela, on réécrit l'arbre d'exécution d'une requête SQL en utilisant les vues dans la base de données. D'une manière générale, cette technique peut être utilisée à chaque fois qu'on veut accéder aux données de la base en utilisant l'interface fournie par une vue. Pour plus d'information, on pourra se référer à [Hal01] qui présente un état de l'art complet sur l'utilisation de vues pour répondre à des requêtes.

Exemple Nous allons illustrer comment la réécriture permet d'optimiser des requête en utilisant des vues matérialisées. Considérons une requête qui cherche les numéros isbn des livres écrits par au moins un auteur français. Cette requête doit sélectionner dans la table `Auteurs` les auteurs français et effectuer une jointure avec la table `Livres`. Cette requête peut s'écrire :

```
SELECT isbn
FROM   Auteurs, Ecrit
WHERE  Auteurs.id = Ecrit.id and pays = "France"
```

Considérons que la vue `Auteurs_F` soit matérialisée. Dans ce cas, la sélection des auteurs français a déjà été calculée. Pour profiter des données stockées accélérer et le traitement de la requête, on peut réécrire la requête en utilisant cette vue :

```
SELECT isbn
FROM   Auteurs_F, Ecrit
WHERE  Auteurs_F.id = Ecrit.id
```

Maintenance de vues matérialisées

La matérialisation des vues permet d'accélérer le traitement des requêtes. L'inconvénient de cette technique, c'est que les données qui sont matérialisées pourront être rendues obsolètes par les modifications des données de la source. En effet, lorsque les sources sont modifiées, le résultat de la vue peut être affecté par cette modification. La **maintenance de vues matérialisées** est le problème visant à maintenir une cohérence entre les données des sources et la matérialisation des vues. Pour cela, le résultat de la vue doit être mis à jour lorsque les données des sources sont modifiées. Il existe principalement deux façons techniques pour cela :

1. recalculer le résultat de la vue,
2. propager d'une façon incrémentale les modifications des sources sur la matérialisation.

La première stratégie peut être qualifiée de naïve et ne demande pas d'efforts particuliers pour être mise en œuvre. Par contre, elle est peu performante car le résultat de la vue est recalculé entièrement à chaque modification.

La seconde stratégie est appelée maintenance incrémentale de vues matérialisées. Elle est plus performante car elle permet de recalculer seulement la partie du résultat de la vue qui a été modifiée. De nombreux travaux ont étudié cette technique, par exemple on peut citer [ZGMHW95, BLT86, CW91, GMS93].

Exemple Considérons la vue `Auteurs_F` définie précédemment sur la base de données. Si l'on ajoute une ligne `(123, "Baril", "Xavier", "France")` dans la table `Auteurs`, le résultat de la vue est affecté. Pour maintenir la cohérence des données de la vue, on devra ajouter la ligne `(123, "Baril", "France")` dans son résultat.

Enfin, l'ajout de méta données peut permettre à une vue matérialisée de se mettre à jour de façon autonome, sans accéder aux données de la source. Ce problème est appelé auto-maintenance, il a été traité dans [QGMW96]. Il peut être difficile à résoudre lorsque la définition des vues à maintenir devient complexe.

Mise à jour des données à travers les vues

Les utilisateurs peuvent éprouver le besoin d'utiliser les vues pour mettre à jour la base de données. Les mises à jour qui sont effectuées sur les vues doivent alors être propagées à la base de données.

Certains attributs du résultat d'une vue ne peuvent pas être modifiés, par exemple les attributs calculés. Considérons par exemple un attribut calculé qui représente la moyenne d'un ensemble de valeurs. Il semble difficile de déterminer quelles valeurs devront être modifiées si l'on veut modifier la moyenne. Pour les autres attributs appartenant au résultat de la vue, il est généralement possible de déterminer de quelle table ils proviennent et de propager la mise à jour.

Exemple Considérons la vue `Auteurs_F` définie précédemment sur la base de données. Si un utilisateur modifie le prénom d'un auteur dans le résultat de la vue, alors le système doit propager cette modification dans la table `Auteurs`. Le système utilise pour cela l'attribut `id` qui permet d'identifier la ligne dans la table et dans la vue.

Cependant, tous les SGBD relationnel commerciaux ne proposent pas ces fonctionnalités. Généralement, seules les vues définies par de simples projections peuvent être modifiées par l'utilisateur. Le résultat des vues plus complexes est verrouillé pour éviter les problèmes de propagation des modifications.

2.2.2 Vues dans le modèle objet

Les SGBD basés sur le modèle objet sont apparus dans les années 90 et une extension du langage SQL a été proposé pour leur interrogation : OQL. Pour les même raisons que dans les SGBD relationnels, des mécanismes de vues ont été proposés. Dans le modèle objet, une vue est généralement définie comme une **classe virtuelle**. Une classe virtuelle est une classe définie par une requête, souvent en utilisant le langage OQL. Le type de la classe est inféré par la requête, tandis que l'extension de la classe correspond au résultat de la requête. Un objet virtuel est une instance d'une classe virtuelle qui possède son propre identifiant.

Pour plus de détails on peut se référer aux travaux du domaine, par exemple [Bel00, AB91, BR94].

Pour des raisons de compatibilité ascendante, les mécanismes de vues pour les SGBD à objets doivent proposer au moins les mêmes fonctionnalités que dans le contexte relationnel (restructuration et intégration de données, réécriture de requête, maintenance incrémentale, mise à jour à travers les vues). Toutefois, le modèle objet a introduit de nouvelles difficultés.

La notion d'identifiant (`oid`), essentielle dans le modèle objet, est la source de nouveaux problèmes. Les identifiants permettent de créer des liens entre les objets. Lorsqu'on veut matérialiser une vue, c'est à dire le résultat de l'exécution d'une requête, on doit gérer les problèmes introduits par les identifiants :

- Les objets de la vue sont-ils de nouveaux objets (dans ce cas on doit générer de nouveaux identifiants) ?
- Comment traiter les objets liés à ceux appartenant au résultat de la vue ?

Le problème de la mise à jour à travers les vues définies par des requêtes multi-classes dans le contexte objet est souvent indécidable : en effet, il n'est pas toujours possible de connaître d'où

viennent les données d'un objet virtuel. Une classification des liens entre les objets à été proposée dans [Bel00], permettant de définir la sémantique à utiliser pour la propagation des modifications. Cette classification permet également de définir des objets pour lesquels les modifications ne seront pas propagées dans la base.

2.3 Intégration de données

2.3.1 Nécessité d'un schéma médiateur

L'intégration **point à point** est une approche possible pour combiner plusieurs sources de données [Gar02, GMUW02]. Cette technique d'intégration consiste à écrire des traducteurs qui implémentent des passerelles entre les bases de données à intégrer. Ces traducteurs autorisent une base de données à interroger une autre base de données de telle façon que celle-ci comprenne la requête. L'inconvénient de cette architecture, c'est l'explosion combinatoire du nombre de connexions à mettre en place si le nombre de bases de données à intégrer augmente.

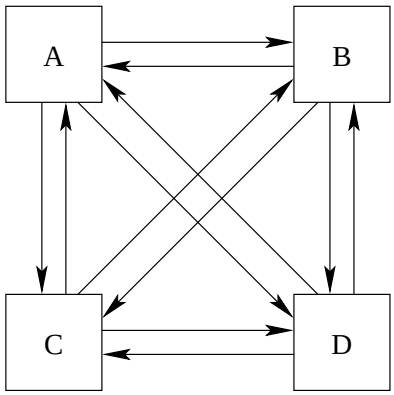

FIG. 2.4 – Quatre sources de données inter-connectées.

La Figure 2.4 présente un système intégrant quatre sources de données avec la technique point à point. Chaque base est connectée aux 3 autres, on voit sur la figure qu'il faut $4(4-1) = 12$ passerelles pour que toutes les bases soient inter-connectées. Les passerelles à implémenter sont donc nombreuses. En effet, pour un système devant intégrer n bases de données, chaque base doit communiquer avec $n-1$ bases. Le nombre total de passerelles à implémenter est alors $n(n-1)$, c'est à dire qu'il faut développer $n(n-1)$ programmes pour permettre les requêtes entre les différentes bases.

On voit donc l'intérêt d'un système d'intégration où un **schéma médiateur** (également appelé schéma global) permet d'intégrer les données provenant de sources multiples. Le nombre de connexions à établir sera égal au nombre de sources à intégrer. En effet, chaque source devra communiquer avec

le système d'intégration. Le choix d'un modèle commun pour le schéma médiateur s'avère également judicieux pour les mêmes raisons.

Définition 2.6 (Système d'intégration de données) *Un système d'intégration de données fournit une vue unifiée de données provenant de sources multiples et hétérogènes. Il permet d'accéder à ces données au travers d'une interface uniforme, sans se soucier de leur structure ni de leur localisation.*

L'interface qui permet de présenter une vue unifiée de données hétérogènes est le schéma médiateur du système. La localisation dans les sources des données qui correspondent à ce schéma doit être transparente pour l'utilisateur du système.

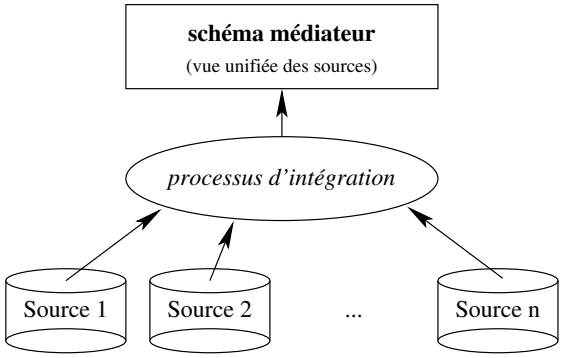

FIG. 2.5 – Système d'intégration de données.

La Figure 2.5 présente une vue synthétique d'un système d'intégration pour illustrer cette définition. Le schéma médiateur du système est construit à partir d'un **processus d'intégration**. Il existe principalement deux méthodes pour intégrer des données : les approches *GAV* et *LAV* que nous présentons plus loin.

La diversité et l'hétérogénéité des sources impliquent de résoudre des conflits pour intégrer les données : on parle de **conflits d'intégration**. Il existe principalement deux types de conflits à résoudre pour intégrer des données provenant de sources multiples et hétérogènes :

1. les conflits **structurels**,
2. les conflits **sémantiques**.

Nous allons présenter une classification détaillée des conflits qui peuvent se présenter lorsqu'on intègre des sources de données hétérogènes. Cette classification qui a été proposée dans [PBE95] permet d'illustrer les différents conflits possibles.

1. Conflits d'identité : le même concept est représenté par différentes entités dans plusieurs sources.
2. Conflits de schéma : on peut distinguer deux sortes de conflits.
   – Les conflits de nom, lorsque le même nom est utilisé pour des concepts différents (homonymie) ou lorsque le même concept est décrit par des noms différents (synonymie).

   - Les conflits structurels, lorsque le même concept est décrit par différents mécanismes.
3. Conflits sémantiques : le même concept est interprété différemment dans les différentes sources.
4. Conflits de données : le même concept est représenté avec des valeurs différentes selon les sources.

Exemple Considérons deux sources de données A et B contenant chacune des données bibliographiques sur des auteurs. Nous allons illustrer les différents types de conflit qui pourraient se présenter pour intégrer ces deux sources. Le processus d'intégration doit fournir une vue C (matérialisée ou virtuelle) proposant un schéma unique et contenant des données provenant des sources A et B.

Un **conflit d'identité** classique est d'avoir le même auteur enregistré dans les deux sources de données A et B. Dans ce cas, il ne doit apparaître qu'une seule fois dans la vue C.

Nous avons vu que les **conflits de schéma** se déclinaient en deux catégories. Nous allons tout d'abord illustrer les **conflits de nom**. Si les auteurs des sources A et B possèdent tous un attribut `ville`, mais que cet attribut contient la ville de résidence pour les auteurs de la source A et la ville de naissance pour ceux de la source B, alors on est en présence d'un conflit d'**homonymie**. En revanche, si les noms de famille des auteurs sont représentés par un attribut `nom` dans la source A et `name` dans la source B, alors on est en présence d'un conflit de **synonymie**.

Les **conflits structurels** sont liés au choix de modélisation et/ou au modèle de données des sources de données. Si la source A est une base de données relationnelle, les auteurs seront représentés par des tuples dans une table. Si la source B est une base de données objet, les auteurs seront représentés par des objets. On sera en présence d'un conflit structurel, car le concept d'auteur sera représenté dans des modèles de données différents.

Certains conflits sont souvent difficiles à résoudre. Si une personne est enregistrée comme auteur dans la source A et comme éditeur dans la source B, alors on a un **conflit sémantique**.

Si un auteur est enregistré dans les deux sources et que son adresse dans la source A est différente de celle dans B, on est en présence d'un **conflit de données**. Pour résoudre ce type de conflit, on introduit souvent un indice de confiance sur les sources de données.

2.3.2 Construction du schéma médiateur : *GAV* et *LAV*

Il existe principalement deux approches pour construire le schéma médiateur d'un système d'intégration de données : les approches *GAV* (*Global As View*) et *LAV* (*Local As View*) [Hal03, MFK01, Vel02]. Ces deux approches sont illustrées par la Figure 2.6.

Avec l'approche *Global as View*, le schéma médiateur est défini comme un ensemble de vues sur les sources. Dans la Figure 2.6(a), la flèche indique que se sont les vues qui définissent le schéma médiateur.

A l'inverse, avec l'approche *Local as View*, le schéma médiateur est défini indépendamment des sources. Les sources sont définies comme des vues sur le schéma médiateur. Dans la Figure 2.6(b), les flèches indiquent que le schéma médiateur est défini sur des vues.

Ces deux approches présentent chacune des points forts et des points faibles. L'interrogation des données des sources à travers le schéma médiateur est plus facile avec *GAV*, tandis que l'ajout de

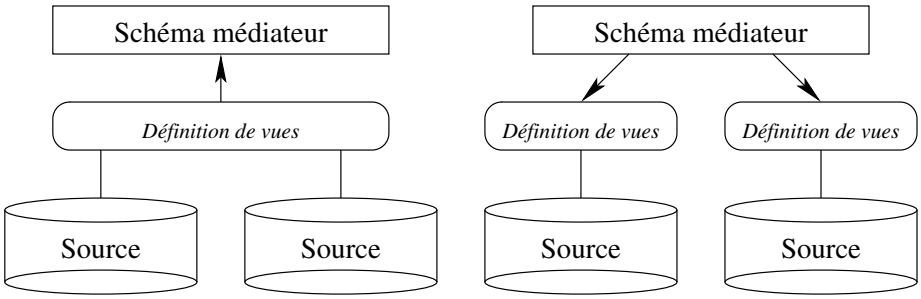

FIG. 2.6 – Approches GAV et LAV.

nouvelles sources est plus facile avec LAV.

L'interrogation d'un système d'intégration se fait généralement en formulant des requêtes sur le schéma médiateur. Le traitement de ces requêtes est plus facile avec l'approche GAV. En effet, la traduction de la requête utilisateur (formulée sur le schéma médiateur) sur le sources peut se faire de manière directe. Un panorama de techniques de réécriture de requêtes en utilisant les vues est présenté dans [Hal01].

Dans le cas de l'approche LAV, ce traitement est plus difficile car le processus de réécriture des requêtes est plus complexe. En effet, il faut utiliser des méta données qui permettent de localiser les vues permettant de répondre à la requête. De plus, la construction du résultat est plus difficile, car il faut recomposer les résultats des requêtes envoyées aux vues qui définissent les sources.

L'ajout de nouvelles sources dans un système d'intégration est plus facile avec l'approche LAV. En effet, il suffit pour cela de spécifier une vue qui permet d'intégrer la nouvelle source dans le schéma médiateur. Cet ajout ne nécessite donc pas de modifier le schéma médiateur. A l'inverse, avec l'approche GAV, l'ajout d'une nouvelle source se fait en modifiant les vues qui définissent le schéma médiateur. Cet ajout peut donc entraîner une modification du schéma médiateur car les vues qui le définissent sont modifiées.

Exemple La Figure 2.7 illustre la construction d'un schéma médiateur en utilisant les approches GAV (a) et LAV (b) dans le contexte des bases de données relationnelles.

Les deux sources à intégrer contiennent des informations sur des auteurs. Elles sont hétérogènes car leurs schémas sont dans des langues différentes. Pour simplifier notre exemple, le schéma médiateur n'est constitué que d'une seule table. Cette table contient des informations sur les auteurs français : `Auteurs_F(nom, prenom)`. Avec l'approche GAV (a), c'est la spécification de la vue qui définit ce schéma médiateur. Avec l'approche LAV (b), une chaque source est définie par une vue sur le schéma médiateur.

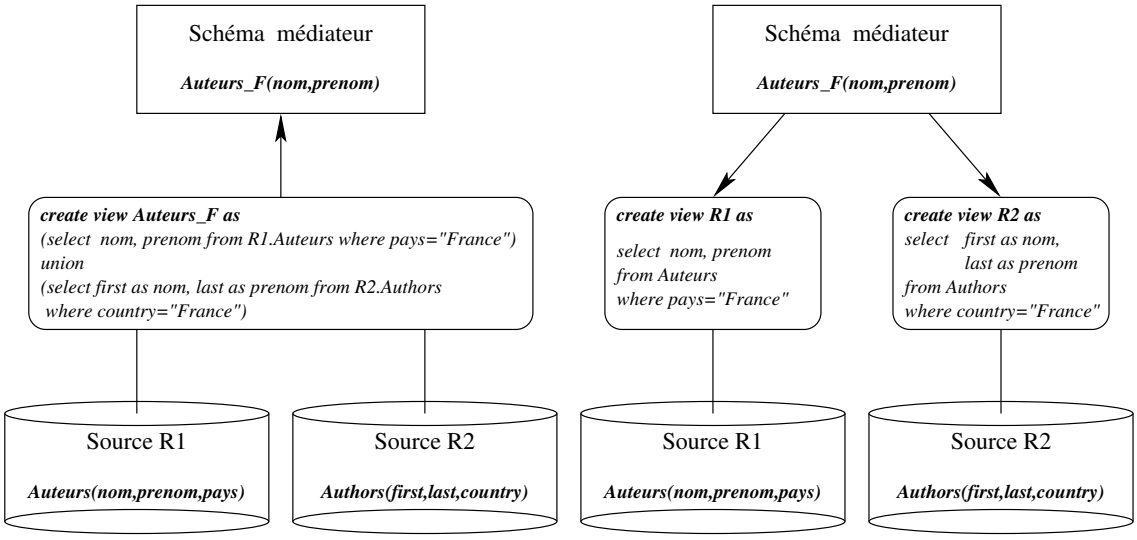

FIG. 2.7 – Exemples d'approches *GAV* et *LAV*.

2.3.3 Architecture

Il existe principalement deux architectures physiques possibles pour les systèmes d'intégration de données. Lorsque les données des sources sont stockées dans le système d'intégration on parle d'approche **matérialisée** (entrepôt de données). A l'inverse, lorsque les données intégrées ne sont pas répliquées, on parle d'approche **virtuelle** (système médiateur). La Figure 2.8 présente ces deux architectures en positionnant parallèlement un **médiateur** et un **entrepôt** pour intégrer des données.

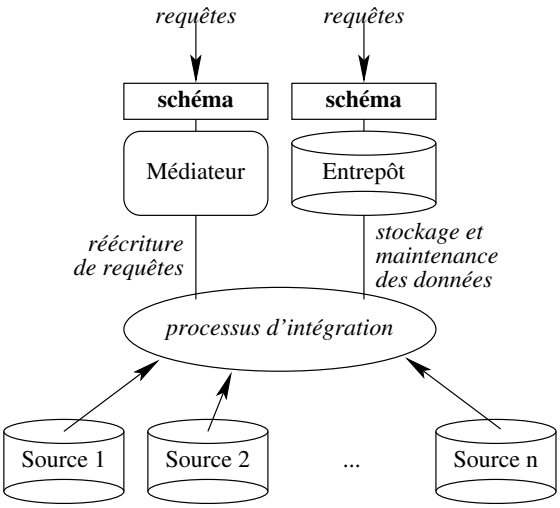

FIG. 2.8 – Architecture d'un système d'intégration.

Système médiateur

Le concept de **médiateur** dans le domaine des systèmes d'information est apparu pour intégrer des sources d'informations hétérogènes. Gio Wiederhold a proposé une architecture logicielle à trois niveaux et a défini un médiateur de la façon suivante : "*un **médiateur** est un module logiciel qui exploite la connaissance de certains ensembles ou sous-ensembles de données pour créer de l'information pour des applications à un niveau supérieur*" [Gio92]. Cette architecture, qui c'est largement imposée dans les systèmes d'information, repose sur les 3 couches suivantes.

1. **Source de données** : cette couche contient les multiples sources qui fournissent les données de base du système.
2. **Médiation** : cette couche contient un médiateur ou une hiérarchie de médiateurs qui permettent de fournir de l'information.
3. **Application** : cette couche contient les applications qui permettent au client d'interroger le système.

Bien que cette architecture en couche ait été définie pour les systèmes médiateurs, elle est également utilisée pour les entrepôts de données qui utilisent une approche matérialisée. La couche de médiation décrite par cette architecture est réalisée par le processus d'intégration, qui peut reposer sur une approche GAV ou LAV.

Le résultat des vues qui permettent d'intégrer les sources n'est pas stocké dans le système médiateur, les données des sources ne sont donc pas répliquées. C'est pour cela qu'on parle d'approche **virtuelle**. Le principal avantage de cette approche, c'est que la mise à jour des données des sources n'a pas d'incidence sur le système. Cela permet d'éviter le problème de la maintenance des données. Le système médiateur doit quand même stocker des méta données sur les sources qu'il intègre. Ces méta données lui permettent de décomposer les requêtes pour interroger ses sources.

La technique utilisée pour répondre aux requêtes est la **réécriture de requêtes**. En effet, lorsqu'une requête est posée au médiateur, elle est décomposée en sous-requêtes qui seront envoyés aux composants qui constituent les sources du médiateur. Comme nous l'avons plus haut, la complexité de cette réécriture dépend fortement du type de l'approche utilisée (GAV ou LAV).

Entrepôts de données

Les données provenant des sources peuvent être stockées dans un **entrepôt de données** [Wid95] après avoir été intégrées. Comme pour les systèmes médiateur, le processus d'intégration utilisé peut être GAV ou LAV. L'entrepôt est construit en **matérialisant** les résultats des vues utilisées pour intégrer les sources. Cette matérialisation permet d'accélérer le traitement des requêtes car il n'est pas nécessaire d'accéder aux sources pour y répondre. Cependant, comme dans l'approche virtuelle il est nécessaire de réécrire les requêtes (exprimées sur le schéma médiateur) sur les vues matérialisées.

Pour maintenir à jour les données de l'entrepôt, on dispose principalement de trois politiques de rafraîchissement.

1. Les données sont rafraîchies **périodiquement**, en reconstruisant entièrement l'entrepôt.
2. Les données sont rafraîchies **périodiquement**, en exécutant **incrémentalement** les mises à jour qui ont été effectuées sur les sources depuis le dernier rafraîchissement de l'entrepôt.
3. Les données sont rafraîchies **dynamiquement**, en exécutant incrémentalement les mises à jour qui sont effectuées sur les sources en **temps-réel**.

Le choix de la technique à mettre en œuvre pour le rafraîchissement de l'entrepôt doit prendre en compte de nombreux paramètres. Lorsque les données des sources évoluent très fréquemment et qu'il n'est pas possible de les maintenir à jour en temps-réel, on doit définir une périodicité qui permet de stocker dans l'entrepôt des données qui ne sont pas trop obsolètes. Heureusement, pour certains problèmes il n'est pas toujours nécessaire de maintenir en permanence la cohérence entre les données des sources et de l'entrepôt. Par exemple, si un entrepôt stocke les données de vente d'un grand magasin, il n'est pas raisonnable de mettre à jour l'entrepôt pour chaque article vendu. De plus, les informations représentant les ventes de la semaine passée peuvent être suffisantes pour être exploitées par les décideurs. L'administrateur de l'entrepôt peut alors décider que les données seront rafraîchies uniquement chaque semaine. Au contraire, pour des données qui évoluent peu fréquemment, la maintenance incrémentale doit permettre de rafraîchir les données sans perturber l'exploitation de l'entrepôt.

Nous ne présentons pas ici un état de l'art du domaine des entrepôts de données. Pour cela on peut se référer par exemple à [Tes00, BG02]. Les données stockées dans un entrepôt sont généralement historisées, nous ne traitons pas cet aspect dans ce mémoire. Cependant, on peut distinguer principalement deux axes de recherche concernant les vues matérialisées.

1. La **maintenance incrémentale** des vues matérialisées permet de propager les mises à jour des sources dans l'entrepôt. Une taxonomie des problèmes de maintenance incrémentale est présentée dans [GM95]. L'utilisation de vues auxiliaires permet de réaliser l'auto-maintenance [QGMW96]. Cette technique consiste à matérialiser des données pour maintenir une vue sans accéder (ou le moins possible) aux données des sources.
2. Les vues à matérialiser pour la construction d'un entrepôt de données sont appelées **configuration** de l'entrepôt. Cette configuration peut être déterminée un ensemble de requêtes et les fréquences d'interrogation et de mise à jour des données. Ce problème est également appelé **sélection de vues à matérialiser**. Pour cela, on utilise un graphe de matérialisation qui contient la décomposition algébrique des requêtes. Chaque nœud de ce graphe est une vue qui peut être matérialisée. Les approches proposées utilisent des modèles de coût et des heuristiques, pour déterminer la configuration de vues à matérialiser qui minimise le temps d'exécution des requêtes et de maintenance des vues [TS97, GM99]. Parallèlement aux travaux présentés dans ce mémoire, nous avons proposé une heuristique basée sur la notion de niveaux dans le graphe de matérialisation des vues [BB03b].

2.3.4 Nécessité d'un modèle commun flexible : vers les données semistructurées

La construction du schéma médiateur (approche *GAV* et *LAV*) s'effectue en définissant des vues sur des sources de données hétérogènes. Certains problèmes liés à l'hétérogénéité du format des sources (bases de données relationnelles, objets, documents HTML, ...) sont résolus grâce à l'utilisation d'un **modèle commun** (également appelé modèle pivot ou global). Ce modèle doit permettre de représenter des données provenant de sources hétérogènes pour les intégrer dans un schéma médiateur.

Les premiers systèmes d'intégration de données ont utilisé les modèles relationnels ou objets comme modèles communs. Cependant, l'intégration de sources de données peu ou pas structurées est difficile avec ces modèles permettant de représenter des bases de données. Les modèles de données semi-structurées ont été conçus pour représenter facilement des données irrégulières provenant de sources hétérogènes, structurées ou non. Ce sont des modèles flexibles qui permettent de représenter des données irrégulières en mélangeant la structure et les données.

Le projet TSIMMIS (*The Stanford-IBM Manager of Multiple Information Sources*[1]) [CGMH+94] avait pour objectif de développer des outils pour faciliter la construction de systèmes d'intégration. Ces systèmes devaient pouvoir intégrer des sources hétérogènes contenant des données structurées et semistructurées.

Le système TSIMMIS utilise une hiérarchie de médiateurs pour intégrer des sources de données hétérogènes. Le modèle commun utilisé est OEM : chaque source est transformée par un extracteur (*wrapper*) pour fournir des données OEM à la hiérarchie de médiateurs. Partant du constat que l'écriture d'extracteurs et de médiateurs est assez longue, les acteurs du projet ont proposé des générateurs de code. Ces générateurs permettent de créer à partir de règles, des extracteurs et des médiateurs pour intégrer des données semistructurées.

Définition 2.7 (Données semistructurées) *Les données semistructurées sont des données dont la structure est irrégulière, éventuellement incomplète et auto-décrite.*

La structure est **irrégulière** et éventuellement incomplète, c'est à dire que les entités décrivant le même concept n'ont pas obligatoirement la même structure et que certaines propriétés d'une entité peuvent être manquantes. Par exemple, un auteur peut être décrit par un nom et prénom et un autre auteur par un nom, un prénom et son âge. Les deux entités qui représentent ces auteurs ont alors une structure irrégulière.

La structure est **auto-décrite**, c'est à dire que le schéma est contenu dans les données. Plus concrètement, une entité contient la description du concept qu'elle représente.

[1] Tsimmis est aussi un mot yiddish pour désigner un ragoût de légumes "hétérogènes" formant un tout étonnamment savoureux.

OEM (*Object Exchange Model*) est le modèle de données semistructurées qui s'est imposé, devenant un standard *de facto* pour les systèmes d'intégration de données apparus avant la naissance d'XML. Il avait été proposé à l'origine dans TSIMMIS [PGMW95], comme modèle commun pour intégrer des données provenant de sources hétérogènes. Nous allons en faire une brève présentation dans cette section. Les données OEM sont représentées par un graphe, dont les nœuds sont des objets permettant de stocker les données et leur schéma.

Un objet OEM est structuré de la manière suivante :

oid	étiquette	type	valeur

- **oid** : un identifiant,
- **étiquette** : une chaîne de caractères qui décrit ce que l'objet représente,
- **type** : le type de la valeur de l'objet,
- **valeur** : la valeur de l'objet.

Les objets OEM peuvent être **simples** ou **complexes**. Les objets simples ont une valeur atomique et sont de type `integer`, `real`, `string`, etc. Les objets complexes sont composés à partir d'autres objets et peuvent être de type `set` ou `list`.

Exemple Les données OEM de la Figure 2.9 décrivent un auteur. L'objet `o1` a pour valeur un ensemble contenant les trois autres objets `o2`, `o3` et `o4`. Plus simplement, on peut dire que l'objet auteur est composé d'un nom, d'un prénom et d'un age.

```
<o1, 'auteur', set, {o2,o3,o4}>
   <o2, 'nom', string, 'Baril'>
   <o3, 'prenom', string, 'Xavier'>
   <o4, 'age', integer, 28>
```

FIG. 2.9 – Exemple de données OEM : description d'un auteur.

Bien que le modèle OEM contienne le terme "objet", un objet OEM ne possède pas de comportement. En effet, il n'est pas possible de définir de méthodes sur un objet OEM. Le seul concept objet présent dans OEM est la notion d'identifiant (`oid`) qui permet de définir un graphe d'objets.

2.4 Systèmes d'intégration basés sur XML

2.4.1 Apport d'XML pour l'intégration de données

La plupart des systèmes d'intégration de données actuels utilisent XML comme modèle commun [BB01c, Gar02] pour les raisons suivantes. Tout d'abord, XML permet de représenter des données provenant de sources hétérogènes, dont les modèles de données sont différents. Ensuite, XML est un

format largement répandu et accepté par la communauté informatique. De nombreuses sources de données sont disponibles et de nombreuses applications permettent d'exporter leurs données en XML.

Le langage XML permet de représenter des données semistructurées, en intégrant la plupart des concepts des modèles connus [Gar02]. Il permet de composer des hiérarchies de données liées entre elles par des hyperliens [XLi, XPo]. Les relations peuvent être vues comme des instances d'éléments avec des attributs. Les hyperliens correspondent aux associations des modèles entité-association. Avec les schémas qui sont apparus récemment [XML01a], le typage des données élémentaire est possible. Pour cela, de nombreux types sont proposés permettant de décrire de manière précise les données. De plus, il est possible de créer des types complexes avec les constructeurs séquence, choix et tas. Enfin, la structure est contenue dans les données sous la forme de balise et de nom d'attribut.

Aujourd'hui, de nombreux logiciels permettent d'exporter leurs données au format XML. En effet, comme c'est un format d'échange, de nombreux éditeurs de logiciel ont développé des modules qui permettent de transformer des données en XML. Parmi les plus connus, le SGBD Oracle^{TM} propose des extensions pour exporter les données contenues dans des tables relationnelles en XML. Certains logiciels des suites bureautiques actuelles de Microsoft^{TM} permettent aussi d'exporter des données XML.

Enfin, la standardisation d'XML par le W3C et son succès dans l'industrie informatique en font un bon candidat comme langage commun pour un système d'intégration. De plus, [DHW01a] souligne qu'un certain effet de mode tend à associer XML à l'intégration de données.

2.4.2 Présentation de systèmes d'intégration basés sur XML

Ces dernières années, de nombreux systèmes se sont basés sur XML pour intégrer des données [LVPV99, ACFR01, DHW01a, GIFF^{+}99, May01, SC99, GMT02, CS02].

Le système YAT [SC99, Sim00] est basé sur XML pour l'intégration de données. Le langage YATL permet de spécifier les données à intégrer de manière déclarative. De plus, une algèbre XML a été proposée pour développer des techniques d'optimisation de requêtes [CCS00].

Il existe également des travaux dans le domaine de l'ingénierie documentaire [CPC95] qui se sont basés sur XML pour proposer des systèmes intégrant des documents. Par exemple, [CS02] propose des descripteurs unifiés basés sur une syntaxe XML, pour des documents multimédias. Ces descripteurs permettent d'interroger un corpus de documents à partir de requêtes pour XML.

Nous présentons ici un panorama de systèmes basés sur XML pour intégrer des données. Tout d'abord MIX [LVPV99] et Xylème [ACFR01] présentent respectivement des exemples représentatifs d'un système médiateur et d'un entrepôt de données XML. Ensuite, nous présentons quelques systèmes commerciaux.

Un système médiateur : MIX

Le projet MIX (*Mediation of Information Using* XML) avait pour objectif de développer un système pour l'interrogation de sources hétérogènes en utilisant XML [MIX]. Ce projet est le fruit d'une collaboration entre l'*UCSD Database Laboratory* et le groupe *Data Intensive Computing Environment* de *SDSC*. Les acteurs du projet considéraient le *Web* comme une grande base de données distribuée et XML comme son modèle de données. Ils pensaient que dans un futur proche, la majorité des sources de données permettrait d'exporter leurs données au format XML. Cette hypothèse semble se vérifier, car de nombreux logiciels permettent aujourd'hui d'exporter leurs données en XML et on parle du *Web* XML [MBV03].

La Figure 2.10 illustre l'architecture du système en présentant les différents composants du système.
- L'interface du système est réalisée par le composant `DOM for Virtual XML Doc's`.
- La médiation des données est réalisée par le composant `MIX Mediator`.

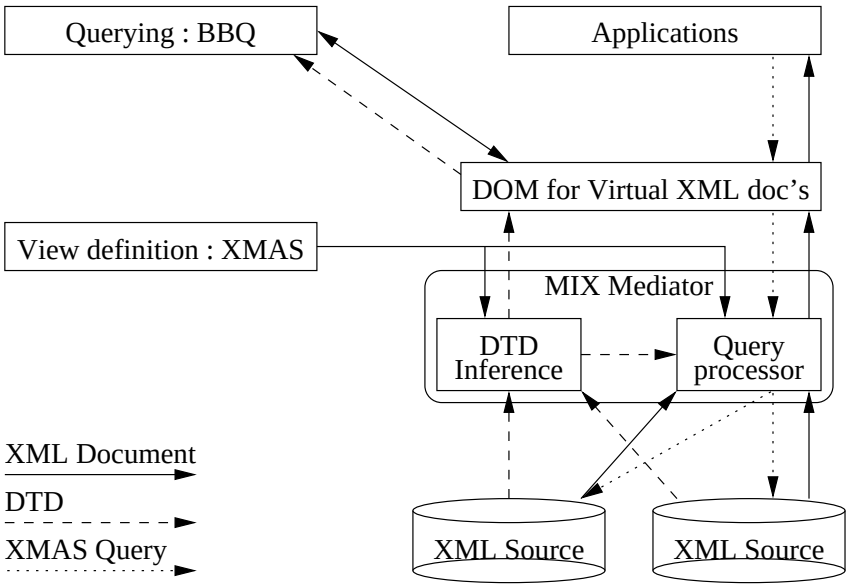

FIG. 2.10 – Architecture du système MIX.

Le schéma médiateur du système est défini par un ensemble de vues sur les sources (*GAV*), à l'aide du langage XMAS [LVPV99]. Ce langage est fortement inspiré de XML-QL [DFF+98], dont il tire l'instanciation de variables par `pattern matching` et la syntaxe. La définition d'une vue permet d'inférer une DTD qui constitue le schéma de la vue.

Le composant `DOM for Virtual XML Doc's` constitue l'interface du système. Il utilise les DTDs générées par les vues pour offrir aux utilisateurs du système des documents XML virtuels. Ces documents sont présentés sous forme d'arbre, en utilisant le modèle de données DOM. Cette interface fournit donc des arbres DOM virtuels qui permettent de fournir aux utilisateurs des documents XML virtuels.

Le composant `MIX Mediator` constitue le cœur du système et permet de construire le schéma médiateur et de traiter les requêtes envoyées à l'interface. Pour cela, il est composé de deux modules.

1. Le module `DTD Inference` est chargé de construire les DTDs à partir de la définition des vues. Ces DTDs permettent à l'interface de fournir le schéma médiateur du système.
2. Le module `Query processor` est chargé de traiter les requêtes. Il décompose les requêtes sur les sources pour les exécuter. Le résultat de l'exécution des requêtes sur les sources est renvoyé à l'interface, pour instancier les documents XML virtuels qu'elle fournit.

Le langage XMAS permet de définir des vues, mais il peut aussi être utilisé pour interroger l'interface du système. Une interface graphique appelée `BBQ` (*Blended Browsing and Querying*) permet de définir des vues ou des requêtes avec XMAS. Cette interface est basée sur les DTDs fournies par les sources (source XML du système pour la définition de vues et document XML virtuel pour la définition de requêtes) pour construire la requête XMAS.

En résumé, MIX est système médiateur utilisant une approche *GAV* virtuelle pour intégrer des données provenant de sources XML multiples et hétérogènes. Le schéma médiateur est construit à l'aide de DTDs qui sont générées par la définition des vues sur les sources. Un interface graphique est également proposée pour la construction de requêtes XMAS, permettant de définir des vues ou d'interroger le système.

Un entrepôt pour intégrer tous les documents XML du *Web* : Xylème

Xylème était à l'origine un projet de recherche [ACFR01, Xylb] qui avait pour objectif de construire un entrepôt de données capable de stocker et de maintenir à jour toutes les données XML du Web. Ce projet qui a impliqué plusieurs centres de recherche (INRIA Rocquencourt, Université de Mannheim, le CNAM de Paris et l'Université de Paris-Orsay) a été achevé en 2000 et a donné naissance à une entreprise commerciale, également appelée Xylème.

Les principaux thèmes de recherche qui ont été étudiés dans le cadre de Xylème peuvent être résumés comme suit :

- stockage efficace de gros volumes de données XML,
- traitement de requêtes avec indexation de documents XML,
- acquisition et maintenance des données XML du Web,
- service de notification de mise à jour de données pertinentes,
- intégration sémantique des données.

Les principaux modules de l'architecture du système sont illustrées dans la figure 2.11. D'un point de vue fonctionnel, on peut découper le système en quatre niveaux :

1. le niveau physique (`Repository and Index Manager`),
2. le niveau logique (`Acquisition & Crawler`, `Loader` et `Query Processor`),
3. le niveau applicatif (`Change Control` et `Semantic Module`),
4. et le niveau interface (`Web Interface` et `Xyleme Interface`).

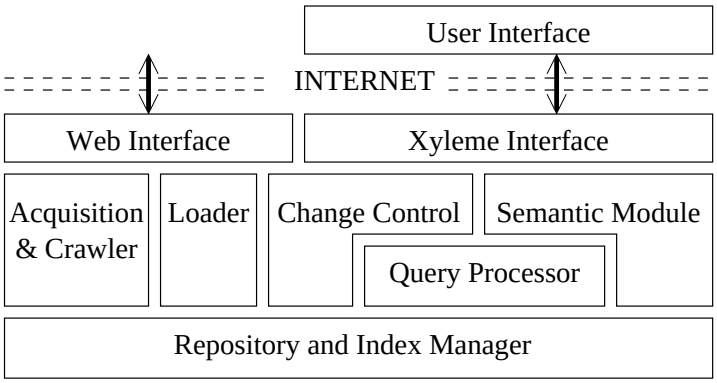

FIG. 2.11 – Architecture fonctionnelle de Xylème.

Stockage Les données sont stockées et indexées dans un SGBD natif dédié à XML : NATIX [KM99]. Ce SGBD a été développé à l'Université de Mannheim et offre des performances qui permettent de traiter de gros volumes de données.

Acquisition et maintenance des données Le système recherche sur le *Web* des documents XML pour peupler l'entrepôt et maintient à jour ces documents [MAA+00]. Pour réaliser ces tâches, des robots sont utilisés. Ces robots parcourent le *Web* à la recherche de documents XML. Pour cela, les pages HTML sont analysées afin de trouver des liens vers des pages XML, ou des pages HTML susceptibles de contenir des liens vers des pages XML. Les données XML extraites sont ensuite stockées dans un entrepôt.

La mise à jour des données se fait aussi à l'aide de robots, qui eux parcourent l'entrepôt. Le rafraîchissement pour maintenir les données à jour (ou le plus possible) se fait principalement selon trois critères que nous présentons brièvement ici.

1. Les pages importantes, c'est à dire qui sont beaucoup référencées, sont rafraîchies plus fréquemment.
2. Les pages changeant peu fréquemment (pages d'archive) sont rafraîchies peu fréquemment.
3. Les pages changeant très souvent (par exemple les cours de bourse qui peuvent changer toutes les minutes), sont rafraîchies peu fréquemment car il est impossible de les maintenir à jour dans un système à grande échelle comme Xylème.

Enfin, des techniques ont été développées pour faire travailler plusieurs robots simultanément.

Système de notification Les utilisateurs de Xylème peuvent être notifiés des mises à jour sur les pages qui les intéressent [NACP01]. Pour cela, un langage de souscription de notification, a été développé pour spécifier des requêtes de surveillance des pages jugées pertinentes par un utilisateur. Des techniques pour détecter des changements dans un document [MACM01] et calculer la différence entre deux documents XML ont été développées [CAM02]. Le système de notification a été conçu et testé pour traiter de grands volumes de documents et de requêtes de notification.

Intégration sémantique Pour interroger l'entrepôt, Xylème propose un mécanisme de vues qui permet de présenter à l'utilisateur une seule structure, qui résume un domaine d'intérêt [RSV01] . En effet, les documents stockés et traitant d'un même domaine peuvent être très nombreux et présenter des structures hétérogènes. Il est alors nécessaire de présenter une vue intégrée des données.

Le schéma médiateur de Xylème est une collection de DTDs **abstraites**. Une DTD abstraite modélise un schéma médiateur pour les données se rapportant à un domaine (par exemple les voyages, la culture, ...). La définition de vues sur les sources s'effectue en générant des *mappings*. Ces *mappings* permettent d'associer les données des sources décrites par des DTDs **concrètes**, aux DTDs abstraites constituant le schéma médiateur. Pour cela, le système effectue un traitement en deux étapes.

1. Construire des DTDs abstraites en classifiant les DTDs concrètes des documents de l'entrepôt. La classification est basée sur une analyse statistique, qui calcule les similarités entre les mots trouvés dans les document et leurs DTDs.
2. Définir les connexions sémantiques, qui permettent d'associer les éléments des DTDs abstraites avec ceux des DTDs concrètes. Les chemins de la DTD abstraite d'un domaine sont alors associés aux chemins des DTDs concrètes des documents de ce domaine.

En résumé, Xylème est un entrepôt de données pour intégrer les documents XML du *Web*. Le schéma médiateur est constitué par une collection de DTDs abstraites. Les vues sont des *mappings* qui associent les chemins des DTDs abstraites aux DTDs concrètes des données des sources. Actuellement, Xylème est une société commerciale [Xyla] qui propose des services *Web* et des produits basés sur le système Xylème (architecture de stockage, interrogation, notification de changement).

Une offre commerciale de système d'intégration de données XML

Nous présentons ici quelques produits commerciaux permettant de construire des systèmes d'intégration de sources de données XML. Ces systèmes construisent le schéma médiateur des données intégrées à l'aide de vues définies avec les langages de requêtes XML proposés par le W3C : XPath [XPa99] et XQuery [XQu03].

E-XMLMedia Cette société [eXm] développe et distribue des composants logiciels, pour faciliter le développement d'application intégrant des sources de données en utilisant XML comme standard d'échange. Les composants s'intègrent dans une architecture distribuée, pour publier, échanger, stocker et interroger des documents XML dans un système d'information [GMT02].

Le composant *XMLizer* joue le rôle d'interface XML pour des données stockées dans un SGBD relationnel. Il permet d'extraire, de transformer et d'insérer des données XML dans un SGBD relationnel en définissant des règles de gestion.

Le composant *e-XML Repository* permet de stocker et d'interroger des documents XML dans un SGBD relationnel. Il étend les fonctionnalités des SGBD relationnels en fournissant un accès dual XML et relationnel aux documents stockés. Il assure un chargement et une restitution rapide des documents stockés dans les tables relationnelles, en conservant et exploitant leur structure pour optimiser leur

placement. Le contenu et la structure des documents sont indexés pour permettre l'interrogation via un langage de requêtes XML. Ce langage de requêtes offre en particulier la possibilité d'interroger simultanément une collection de documents XML. Pour cela, on utilise des expressions XPath permettant de naviguer dans la structure et de sélectionner des fragments des documents XML. Il permet aussi un mode de stockage générique utilisé pour stocker tout type de documents XML, y compris ceux dont la structure n'est pas connue a priori, sans configuration préalable. Lorsque la structure des documents stockés est connue, l'utilisateur peut définir une correspondance entre cette structure (XML S*chema*) et un modèle relationnel.

Le composant *e-XML Mediator* est un outil pour exécuter des requêtes sur des sources de données XML multiples et hétérogènes. La localisation des données dans les sources est transparente, grâce à l'ajout de méta données. En fait, ces méta données constituent des vues locales qui définissent les sources pour les intégrer dans un schéma selon une approche *LAV*. Le composant *e-XML Mediator* se connecte aux sources de données via des *wrappers* (adaptateurs) qui assurent : (i) la traduction des données du format natif de chaque source vers XML, (ii) la traduction de la requête XML vers le langage de requête natif de la source. Des ***wrappers*** génériques permettent d'accéder à des sources SQL, *e-XML Repository* et HTML. Une interface de programmation permet de développer des **wrappers** spécifiques pour d'autres types de sources.

Nimble Cette société propose une plate-forme d'intégration du même nom en utilisant une approche *LAV* [DHW01a, DHW01b]. Son architecture est présentée dans la Figure 2.12.

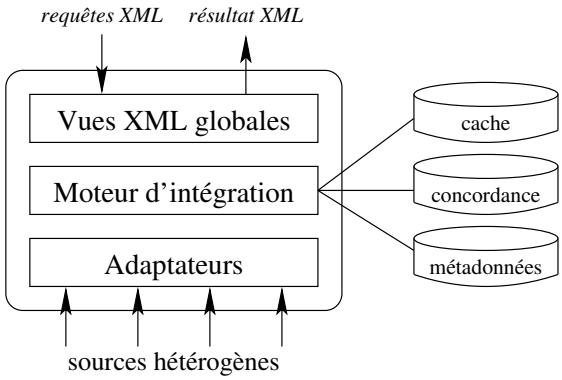

FIG. 2.12 – Architecture du système *Nimble*.

Le processus d'intégration repose sur la définition de cartes XML des sources de données. Une telle carte décrit hiérarchiquement les éléments de données issus d'une source et la correspondance entre le schéma médiateur et les données XML de la source. Ces cartes sont stockées dans la base de concordance qui contient les *mappings* du schéma local d'une source au schéma médiateur. Plus généralement, elle permet de résoudre les conflits de noms des sources de données à intégrer. De plus, certaines vues peuvent être matérialisées dans un cache. Cependant, la maintenance du cache ne propose pas de technique de propagation des mises à jour des sources.

D'autres sociétés proposent des outils pour intégrer des données XML. On peut citer par exemple *Liquid Data* de BEA [BEA] qui permet de définir des vues intégrées de sources de données hétérogènes et de les interroger avec XQuery. *Liquid Data* s'intègre dans le serveur d'application de la société pour construire des portails et développer des services *Web*. La société *Enosys Software* [Eno] propose également un moteur d'intégration de données hétérogènes qui permet d'interroger des vues XML avec XQuery. L'ouvrage [Gar02] présente plus en détail ces deux produits.

2.5 Conclusion et proposition

Les systèmes d'intégration sont basés sur le mécanisme de vues pour présenter une vue unifiée de données provenant de sources hétérogènes. Cette vue unifiée est appelée schéma médiateur du système.

Il existe principalement deux approches pour construire le **schéma médiateur** d'un système d'intégration. L'approche *GAV* définit le schéma médiateur comme une collection de vues sur les sources. A l'inverse, avec l'approche *LAV* les sources de données sont définies comme des vues sur le schéma médiateur du système.

Indépendamment de l'approche utilisée pour la construction du schéma médiateur, deux architectures sont possibles pour construire un système d'intégration. Lorsque les vues sont virtuelles, ont parle de **système médiateur** [Gio92]. A l'inverse, lorsque les vues sont matérialisées, on parle d'**entrepôt de données** [Wid95].

Le langage XML présente les qualités nécessaires pour être utilisé comme **modèle commun** pour intégrer des données provenant de sources hétérogènes. En effet, il permet de représenter des données semistructurées. Il est donc nécessaire de définir un mécanisme de vues pour XML. Nous avons présenté un tour d'horizon de différents systèmes basés sur XML pour intégrer des données provenant de sources hétérogènes.

Notre contribution est de proposer un modèle de vues pour l'intégration de données XML. Le choix d'XML comme modèle commun permet d'intégrer des sources hétérogènes. Cette hypothèse s'appuie sur les raisons suivantes.

1. La transformation de sources de données hétérogènes en XML a été largement étudiée dans la littérature [Gar02, GMT02].
2. De nombreuses sources XML sont disponibles sur le *Web* [MBV03].
3. Certaines applications actuelles permettent d'exporter leurs données en XML.

Notre modèle de vues, VIMIX (VIew Model for Integration of XML sources) repose sur un modèle de données qui permet de représenter les liens de référence dans un document XML. La spécification d'une

vue utilise les concepts de motifs sur les sources pour spécifier les données des sources à extraire pour être intégrées. L'utilisation de fragments (qui calculent l'union) et de jointures permet de restructurer et de combiner les données extraites des sources. Enfin, le résultat de la vue est défini par un arbre qui décrit sa structure. Cet arbre qui contient des variables qui permettent de spécifier les données des sources à utiliser pour générer le résultat de la vue.

Le langage de définition de vues de VIMIX permet d'intégrer des données en utilisant une approche mixte GAV/LAV. En effet, la définition de motifs sur les sources permet de représenter les données extraites des sources selon un schéma défini à l'avance, comme dans une approche LAV. La spécification des vues VIMIX permet de définir le schéma médiateur d'un système d'intégration, comme dans une approche GAV. Pour cela, les opérations d'union et de jointure permettent de restructurer les données extraites des sources. La spécification du résultat d'une vue utilise les données extraites des sources et permet de définir de nouveaux éléments et attributs XML. Des opérations de groupage et d'agrégation sont possibles pour construire ces nouveaux éléments.

Notre modèle de vues VIMIX est présenté dans le chapitre suivant. Le chapitre 3 présente le modèle de données que nous avons utilisé pour XML ainsi que le langage qui permet de spécifier les vues. Le chapitre 4 montre comment VIMIX permet d'intégrer des données provenant de sources hétérogènes et définir un schéma médiateur.

Nous avons également proposé une méthode de stockage qui permet d'utiliser un SGBD relationnel pour matérialiser des vues VIMIX. Cette méthode permet de maintenir les données matérialisées de manière incrémentale, dans le contexte de sources XML qui ne sont pas monitorées.

Chapitre 3

Un modèle de vues pour XML : VIMIX

3.1 Introduction

Dans ce chapitre, nous présentons notre modèle de vues pour XML. Ce modèle, appelé VIMIX (VIew Model for Integration of XML sources) permet de définir des vues sur des sources de données XML. Généralement, un modèle de vues est basé sur un **modèle de données** et un **langage de requêtes**. Il fournit un **langage de définition de vues** dont le pouvoir d'expression correspond au langage de requête utilisé.

Il y a eu de nombreuses propositions de langages de requêtes pour XML [QL'98, RLS98, DFF+98, FS98, CCD+98, CRF00] et une classification a été proposée dans [LM00]. Très récemment, le W3C a proposé un langage qui est un cours de recommandation[1] : XQuery[XQu03]. Ce langage est basé sur *Quilt* [CRF00] et tire profit des qualités présentées par les langages précédents. La localisation des nœuds dans les documents XML s'appuie sur le langage XPath [XPa99]. Le fonctionnement par *pattern-matching* et la possibilité de restructurer les éléments du résultat d'un requête sont largement inspirés de XML-QL [DFF+98]. Enfin, certaines clauses et la construction de variables avec des expressions de navigation ont été empruntées respectivement aux langages SQL et OQL.

Lorsque nous avons débuté nos travaux pour définir notre modèle de vues, il n'existait pas de proposition stable du W3C pour interroger des documents XML. Notre proposition de langage de définition de vues [BB00] s'est inspirée de XML-QL. Nous avons utilisé le *pattern-matching* qui permet de spécifier les données à extraire dans les sources XML. C'est cette partie du langage de définition de vues que nous présentons dans ce chapitre. De plus, comme dans XML-QL notre langage de définition de vues spécifie le résultat en décrivant un arbre XML. Cet arbre utilise des expressions basées sur les variables définies par *pattern-matching* qui permettent de peupler le résultat. Enfin, pour restructurer les données extraites des sources, nous avons proposé des fonctions basées sur SQL pour faire l'union et la jointure des données des sources.

[1] Une recommandation correspond à une normalisation pour le W3C.

Pour la syntaxe du langage de définition de vues nous avons utlisé XML. L'utilisation d'XML pour définir un langage (même manipulant des données XML) a été préconisée dans [Mai98]. Notre choix est motivé principalement par les trois arguments suivants.

1. Comme nous l'avons souligné dans le chapitre 2, XML est adapté pour représenter des données semistructurées. La spécification de vues pour XML est par nature semistructurée : de nombreuses propriétés sont optionnelles. De plus, la structure des données XML permet de représenter naturellement les arbres qui spécifient certains concepts du langage de vues (les axes de recherche et la structure du résultat d'une vue).
2. Ensuite, XML est un format très utilisé pour l'échange de données. Son choix permet d'échanger facilement la spécification de vues entre les différents composants d'une application ou entre différentes applications.
3. Enfin, le choix d'XML facilite l'analyse de la spécification du système d'intégration. En effet, il existe de nombreux outils qui permettent d'extraire la structure des données. Pour cela, on peut utiliser un *parser* qui générera un arbre DOM (ou un graphe de données comme celui défini dans la section 3.2.2) représentant la spécification des vues.

Les principaux objectifs qui ont guidé la conception de notre langage de définition de vues sont au nombre de quatre. Ils concernent (1) la fermeture du langage de définition de vues, les possibilités de (2) restructuration et (3) d'intégration de données et de (4) génération de schéma.

1. **Fermeture du langage**
   Le résultat d'une vue doit être un document XML. La Figure 3.1 illustre la propriété de fermeture du langage de définition de vues. Les sources qui constituent le domaine de la vue sont des documents XML. Le résultat de la vue est également un document XML. Cette propriété permet d'utiliser de manière transparente le résultat d'une vue ou tout autre source de données XML. C'est une propriété importante car elle permet de visualiser et d'interroger de la même manière une vue et une source XML.

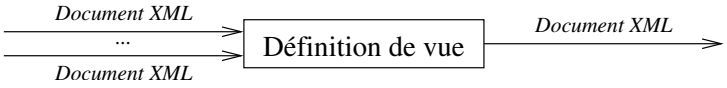

FIG. 3.1 – Fermeture du langage de définition de vues.

   Cet objectif est atteint par l'utilisation d'un arbre qui spécifie la forme du résultat comme un arbre XML.

2. **Possibilités de restructurations**
   Le langage de définition de vues doit permettre de restructurer les données des sources XML. On peut distinguer deux sortes de données dans le résultat d'une vue. Les nœuds de données XML qui proviennent directement d'une source et ceux qui ont été construits par la spécification de la vue. Pour construire de nouveaux nœuds de données XML, plusieurs solutions sont possibles.

- On peut utiliser des fonctions d'agrégation (`sum`, `avg`, `min`, `max`, `count`). A l'exception de `count`, les fonctions présentées sont généralement définies sur des domaines de valeurs numériques. Lorsqu'elles sont appliquées à des valeurs dans des documents XML, une coercion de type est appliquée sur les chaînes de caractères, afin de pouvoir utiliser les fonctions numériques.
- On peut construire de nouveaux éléments en utilisant des données provenant d'éléments différents dans la (les) source(s) utilisée(s) par la vue. De plus, le langage doit proposer un mécanisme de jointure qui permet d'associer des données provenant de sources différentes.

Ces objectifs sont atteints par la spécification de la forme du résultat : elle permet de spécifier de nouveaux éléments et attributs. De plus, l'utilisation d'expressions permet de transformer les nœuds des sources de données.

3. **Possibilités pour l'intégration de données**
   Le langage de définition de vues doit permettre de résoudre certains conflits entre les données des sources. Le choix d'XML permet de résoudre les problèmes syntaxiques de l'intégration. Cependant, il faut résoudre les problèmes liés à l'hétérogénéité de la structure des différentes sources à intégrer. De plus, il peut y avoir des conflits de données entre les différentes sources (données redondantes). Pour intégrer des données, notre langage de définition de vues devra offrir les possibilités suivantes :
   - Faire l'union de plusieurs sources,
   - Faire la jointure de plusieurs sources,
   - Eliminer les redondances provenant de sources multiples.

   Ces objectifs sont atteints par l'utilisation de motifs, de fragments et de jointures pour spécifier les données d'une vue. Nous évoquerons plus en détail comment notre modèle de vues permet d'intégrer des données dans le chapitre suivant.
4. **Inférence de** DTD
   Le schéma du résultat d'une vue est spécifié par la définition de la vue. Dans le processus d'intégration de données, les schémas des vues permet de définir un schéma médiateur. Notre langage de définition de vues doit permettre de générer ce schéma. Dans le contexte XML, les DTDs permettent de valider des documents, on peut les considérer comme des schémas pour les documents. Comme le résultat d'une vue est un document XML, notre langage doit permettre de générer sa DTD.

Ce chapitre est organisé comme suit.

- Dans la section 3.2, nous présentons le modèle de données que nous avons défini pour manipuler des sources XML. Notre modèle permet de représenter les données XML dans un graphe. Nous prenons en compte les liens de composition et de référence entre les éléments et les attributs.
- Dans la section 3.3, nous présentons la partie de notre langage de définition de vues qui permet de spécifier les données d'une source à extraire. Pour cela, nous utilisons des motifs sur les sources (*source-pattern*) qui permettent de lier les données des sources à des variables. D'un point de vue logique, les données extraites par un motif peuvent être représentées par une table relationnelle.

– Dans la section 3.4, nous présentons la partie de notre langage de vues qui permet de définir l'union de plusieurs sources de données. Pour cela, nous avons défini des fragments qui permettent de faire l'union de données extraites des sources. D'un point de vue logique, les données spécifiées par un fragment peuvent être représentées par une table relationnelle.
– Dans la section 3.5, nous présentons la partie de notre langage de vues qui permet de définir la jointure de plusieurs sources de données. Pour cela, nous avons défini des jointures qui permettent de faire le croisement de données extraites des sources. D'un point de vue logique, les données spécifiées par une jointure peuvent être représentées par une table relationnelle.
– Dans la section 3.6, nous présentons la partie de notre langage de définition de vues qui permet de spécifier des vues intégrant des données XML. Les données utilisées par la vue sont spécifiées en utilisant une source de données (motif, fragment ou jointure) définie dans les sections précédentes. Pour spécifier la forme du résultat de la vue, nous utilisons un arbre qui définit la structure des données du résultat. La spécification de la vue permet de générer un schéma représenté par une DTD.

3.2 Un modèle de données pour XML

Dans cette section, nous présentons le modèle de données que nous avons utilisé pour représenter des données XML. Les documents XML sont structurés en arbres d'éléments. Les structures d'arbre ou de graphe ont donc été largement utilisées pour représenter leurs données [XPa99, FS98, DFF+98, QL'98, Bru01, Vel02, FK99]. Comme la plupart des propositions, notre modèle de données permet de différencier les éléments et les attributs. Pour cela, nous distinguons trois types de nœuds (élément, attribut et texte), comme dans le modèle de données proposé pour XPath [XPa99]. De plus, nous proposons deux types de liens pour notre graphe de données, ce qui permet de représenter les liens de référence proposés par le langage XML à travers le mécanisme d'attributs (`ID/IREF(S)`).

3.2.1 Caractéristiques générales

Notion d'ordre

Dans un document XML, les éléments et les attributs sont ordonnés par leur ordre d'apparition. Dans la norme du langage, seul l'ordre des éléments est pris en compte : l'ordre des attributs n'a pas d'importance. Les deux éléments XML `<elt a1 a2 />` et `<elt a2 a1 />` sont donc équivalents. Certains modèles de données (XML *Tree* dans Xylème [Vel02]) ne prennent en compte ni l'ordre des éléments, ni celui des attributs afin de simplifier la représentation.

Nous avons choisi de prendre en compte l'ordre des éléments et des attributs. Cet ordre nous semble important car il peut être utilisé par l'auteur d'un document XML pour représenter de la sémantique. Considérons par exemple un élément `personne` qui possède deux attributs (ou éléments) `tel-fixe` et `tel-portable`. L'ordre d'apparition des attributs (ou des éléments) peut avoir une sémantique : l'attribut (ou l'élément) qui apparaît le premier est le numéro de téléphone principal de la personne.

De plus, concernant les éléments, l'ordre d'apparition permet de distinguer plusieurs éléments frères qui sont de même type. Par exemple, si un élément `personne` possède deux sous-éléments `adresse`, leur ordre d'apparition permet de les distinguer.

Partage d'éléments

Les DTDs permettent de définir des attributs de type `ID` pour identifier certains éléments. Les attributs de type `IDREF(S)` permettent de faire référence à un (ou plusieurs) élément(s). Ce mécanisme permet de partager des éléments à l'intérieur d'un document.

L'imbrication des éléments permet de représenter un document comme un arbre d'éléments. Le mécanisme de partage des éléments, si on veut le prendre en compte, nécessite de représenter les données du document par un graphe. Ce graphe comporte un nœud racine qui représente l'élément racine du document.

On peut donc distinguer deux types d'arcs dans le graphe représentant les données d'un document XML.

- Les liens de **composition** permettent de représenter l'imbrication des éléments.
- Les liens de **référence** permettent de représenter le mécanisme de partage des éléments.

Nous avons choisi de prendre en compte les liens de référence dans notre modèle de données, car ils permettent de traduire la sémantique ajoutée par les DTDs grâce aux attributs `ID` et `IDREF(S)`. Le modèle de données utilisé dans Xylème (XML *Tree*) et l'interface de programmation DOM ne permettent pas cette fonctionnalité.

3.2.2 Graphe de données XML

Notre modèle de données consiste en un graphe $G(N, E, n)$ qui permet de représenter les données d'un document XML. Ce graphe posséde les caractéristiques suivantes :

- N est un ensemble contenant tous les nœuds du document,
- E est un ensemble contenant tous les arcs du document,
- n est le nœud racine de G ($n \in N$) représentant la racine du document.

On distingue trois types de nœuds :

- *Element* : permet de représenter les éléments XML,
- *Attribut* : permet de représenter les attributs des éléments,
- *Text* : permet de représenter les chaînes de caractères qui constituent le contenu du document.

On distingue deux types d'arcs :

- *Composition* : permettent de représenter les liens de composition,
- *Reference* : permettent de représenter les liens de référence.

La Figure 3.2 illustre une DTD validant des données bibliographiques. Un exemple de document validé par cette DTD est donné en annexe (B.1, page 155).

```
<?xml version="1.0" encoding="ISO8859_1" ?>

<!-- element racine -->
<!ELEMENT bibliographie (auteur+, publication*) >

<!-- auteurs -->
<!ELEMENT auteur (nom, prenom, email?, web?) >
<!ATTLIST auteur
          id ID #REQUIRED
>

<!-- publications -->
<!ELEMENT publication (titre, type, annee, titre-livre, pages, isbn?) >
<!ATTLIST publication
          id ID #REQUIRED
          auteurs IDREFS #REQUIRED
>

<!-- PCDATA -->
<!ELEMENT nom    (#PCDATA) >
<!ELEMENT prenom (#PCDATA) >
<!ELEMENT email  (#PCDATA) >
<!ELEMENT web    (#PCDATA) >

<!ELEMENT titre       (#PCDATA) >
<!ELEMENT type        (#PCDATA) >
<!ELEMENT annee       (#PCDATA) >
<!ELEMENT titre-livre (#PCDATA) >
<!ELEMENT pages       (#PCDATA) >
<!ELEMENT isbn        (#PCDATA) >
```

FIG. 3.2 – DTD validant des données bibliographiques.

La racine du document est un élément `bibliographie` qui doit être composé d'un ou plusieurs éléments `auteur` et de zéro ou plusieurs éléments `publication`. Les éléments `auteur` possèdent un attribut `id` de type `ID` qui permet de les identifier. Ils sont composés obligatoirement des sous-éléments `nom` et `prenom` et éventuellement des sous-éléments `email` et `web`. Les publications sont, comme les auteurs, identifiées par un attribut `id`. L'attribut `auteurs` (de type `IDREFS`) permet de référencer les auteurs d'une publication. Enfin, chaque élément `publication` est composé des sous-éléments `titre`, `type` (indiquant si la publication est une revue, une conférence, etc.), `annee`, `titre-livre` (indiquant le titre de la revue ou de la conférence) et `pages`.

La Figure 3.3 contient un graphe XML représentant des données bibliographiques qui respectent la DTD décrite précédemment (Figure 3.2). Le document contenant les données correspondantes est présenté en annexe (B.1, page 155). Pour des contraintes d'espace, nous n'avons pas représenté toutes les données du document dans le graphe.

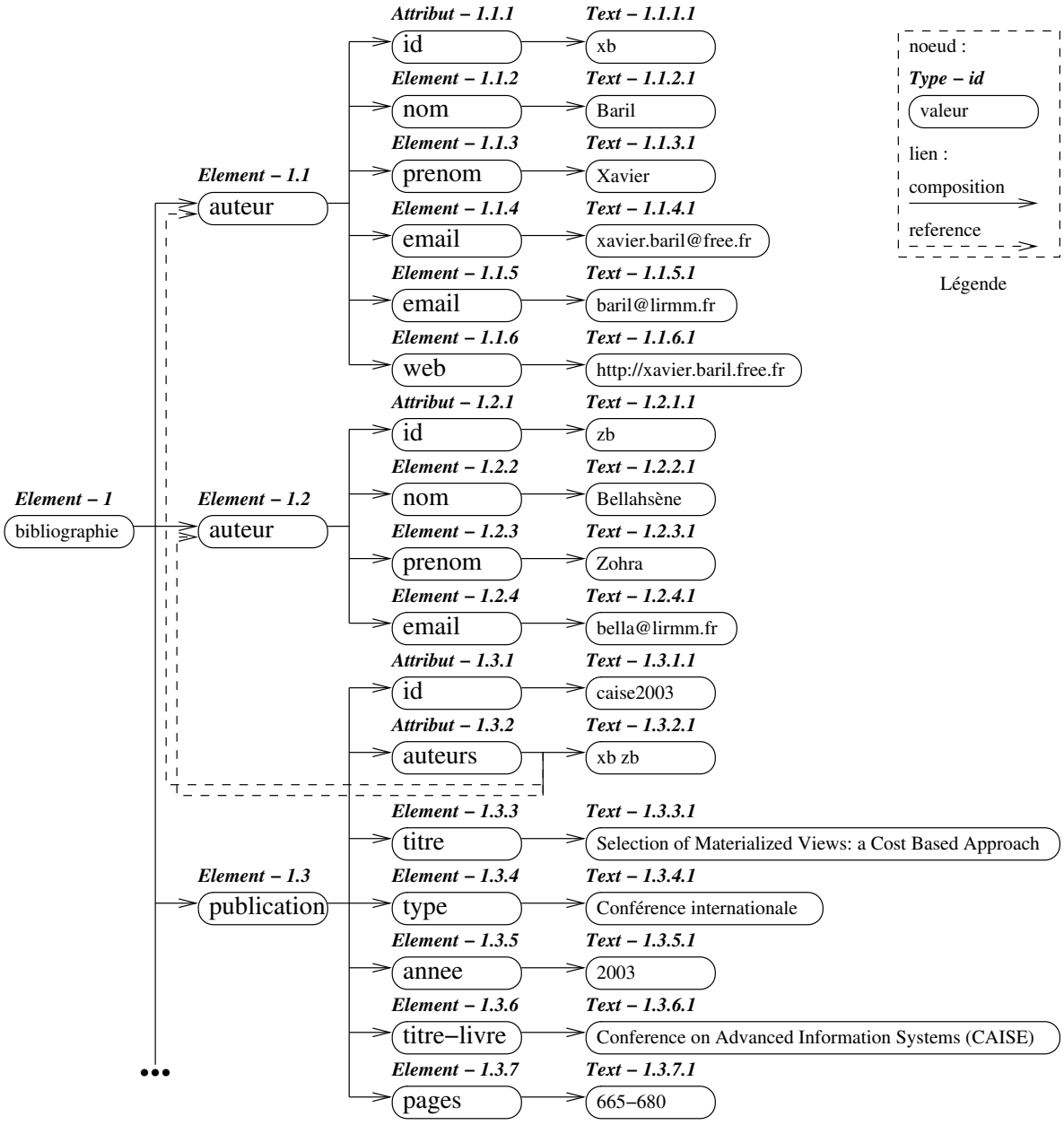

FIG. 3.3 – Graphe XML représentant des données bibliographiques.

Le graphe de données XML de la Figure 3.3 est représenté avec le formalisme suivant :

- les nœuds sont dans des boites avec les bords arrondis,
- la valeur du nœud est en caractères standards,
- les liens de composition sont en traits pleins,
- les liens de référence sont en traits pointillés,
- l'ordre des fils est donné par l'ordre de départ des flèches (de haut en bas),
- le type du nœud est en caractère gras italique (en haut à gauche de la boite).

Remarque Notre modèle de données permet également de représenter des documents ayant des contenus mixtes. Un contenu est dit "mixte" s'il contient à un même niveau hiérarchique des nœuds éléments et textuels. Considérons par exemple l'élément suivant :

```
<publication>
<titre>A View Model for XML Documents</titre>, In proceedings of
<conference>OOIS'<annee>2000</annee></conference>
</publication>
```

La Figure 3.4 illustre le graphe XML représentant ce contenu mixte.

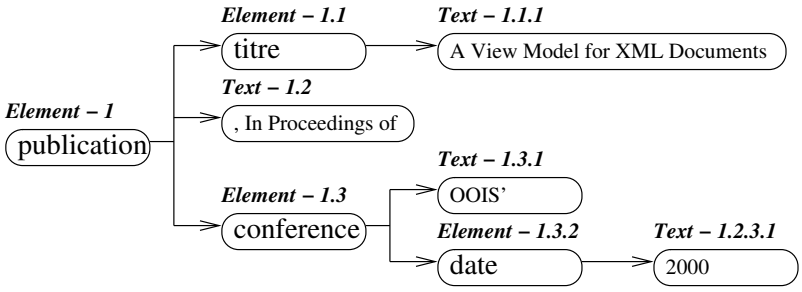

FIG. 3.4 – Graphe XML représentant un contenu mixte.

3.2.3 Opérations sur les nœuds du graphe

Nous proposons des opérations qui permettent de manipuler les nœuds du graphe. Ces opérations peuvent être classées en trois catégories :

- test sur le type de nœud,
- manipulation de chaînes de caractères,
- navigation dans le graphe.

Tests sur les nœuds du graphe

Les fonctions permettant de tester le type d'un nœud renvoient un booléen pour indiquer le résultat. Il y a une fonction pour chaque type de nœud. Les signatures de ces fonctions sont :

- `isAttribute(node) : boolean`
- `isElement(node) : boolean`
- `isText(node) : boolean`

Manipulation de chaînes de caractères

Nous proposons deux fonctions qui permettent de calculer la représentation textuelle d'un nœud. Ces fonctions renvoient chacune une chaîne de caractères contenant cette représentation. Les signatures de ces fonctions sont :

- `text(node) : string`
- `r-text(node) : string`

Le résultat de la fonction `text()` dépend du type du nœud.

- Si le nœud est de type texte, la valeur du nœud est renvoyée.
- Si le nœud est de type attribut, la valeur de son nœud fils texte est renvoyée.
- Si le nœud est de type élément, la valeur de son fils texte est renvoyée. Si le nœud n'a pas de fils texte, alors une chaîne vide est renvoyée.

La fonction `r-text()` a un comportement identique à celui de la fonction `text()` pour les nœuds de type texte et attribut. Par contre, pour un nœud de type élément, la fonction `r-text()` calcule de manière récursive la représentation textuelle du nœud en effectuant un parcours en profondeur d'abord des fils du nœud. Toutes les représentations textuelles des descendants sont séparées par des espaces.

Exemples Considérons le graphe décrit par la Figure 3.3 (les nœuds sont identifiés par leur position).

- `text(1.3) = ""` (car le nœud n'a pas de fils texte),
- `text(1.3.2) = "xb zb"`,
- `text(1.3.4) = "Conférence internationale"`,
- `text(1.3.4.1) = "Conférence internationale"`,
- `r-text(1.1) = "xb Baril Xavier xavier.baril@free.fr baril@lirmm.fr ..."`,

Navigation dans le graphe

Nous proposons des fonctions qui permettent de naviguer dans le graphe à partir d'un nœud. Chaque fonction renvoie la liste de nœuds que l'on peut atteindre, à partir d'un nœud de départ passé en paramètre. Cette liste peut être vide. Les opérations de navigation que nous avons considérées sont `children` qui renvoie tous les fils d'un nœud et `descendant` qui renvoie tous les descendants d'un nœud. Pour chacune de ces opérations, nous avons considéré le graphe complet et le sous graphe composé seulement des liens de composition.

Nous utiliserons la notation "`< >`" pour construire des listes. La notation "`<node>`" représente donc une liste de nœuds. Les signatures de ces fonctions de navigation sont :

- `children(node) : <node>`
  Cette fonction renvoie tous les fils d'un nœud.
- `children-comp() : <node>`
  Cette fonction renvoie tous les fils d'un nœud dans le sous-graphe contenant seulement des liens de composition.
- `descendants() : <node>`
  Cette fonction renvoie tous les descendants d'un nœud en effectuant un parcours en profondeur

d'abord.

– `descendants-comp() : <node>`
  Cette fonction renvoie tous les descendants d'un nœud en effectuant un parcours en profondeur d'abord, dans le sous-graphe contenant seulement des liens de composition.

Exemples Considérons le graphe décrit par la Figure 3.3 où les nœuds sont identifiés par leur position.

– `children(1) = <1.1, 1.2, 1.3>`,
– `children-comp(1) = <1.1, 1.2, 1.3>`,
– `children(1.3.2) = <1.3.2.1, 1.1, 1.2>`,
– `children-comp(1.3.2) = <1.3.2.1>`,
– `children(1.1.1.1) = <>`,
– `descendants(1.3.2) = <1.3.2.1, 1.1, descendants(1.1), 1.2, descendants(1.2)>`,
– `descendants-comp(1.3.2) = <1.3.2.1>`,
– `descendants(1.1) = <1.1.1, 1.1.1.1, 1.1.2, 1.1.2.1, ... 1.1.6, 1.1.6.1>`,
– `descendants(1.2) = <1.2.1, 1.2.1.1, 1.2.2, 1.2.2.1, ... 1.2.4, 1.2.4.1>`.

3.2.4 Remarques sur le graphe de données XML

Nous considérons trois types de nœuds présents dans un document XML : les éléments, les attributs et les chaînes de caractères qui constituent le contenu du document. Notre modèle permet de représenter le mécanisme de partage d'éléments proposé par XML avec les attributs de type `ID` et `IDREF(S)`. Les fonctions de navigation que nous proposons permettent d'utiliser ces liens. A notre connaissance, il n'y a pas d'autre modèle de données qui présente cette fonctionnalité.

Notre graphe de données XML possède deux types de liens. Les liens de composition représentent l'imbrication des éléments. Dans un document XML, tous les éléments sont imbriqués à l'intérieur d'un élément racine : les données sont organisées dans un arbre d'éléments. Ceci nous permet de remarquer que **le sous-graphe composé uniquement de liens de composition est un arbre**. Cette propriété est utilisée pour l'implémentation des fonctions de navigation du modèle de données.

3.3 Données des sources à extraire

La première étape, lorsqu'on veut intégrer des données, est d'extraire les données des sources. Pour cela, notre langage de vues, propose des **motifs sur les sources**. Un motif est appelé *source-pattern* dans notre modèle de vues.

Cette section est organisée de la manière suivante. Tout d'abord nous présentons la partie de VIMIX permettant la spécification d'un motif sur une source (*source-pattern*). Cette spécification est basée sur le concept de *pattern-matching*, qui consiste à spécifier un motif à retrouver dans une source de données pour définir des variables. L'instanciation de ces variables permettra d'extraire les données de cette source. Enfin, nous présentons comment les données à extraire sont représentées d'un point de vue logique.

3.3.1 Définition de motifs sur les sources

La spécification des données à extraire se fait par ***pattern-matching*** : les données des sources sont liées à des variables qui sont déclarées dans un motif décrivant la source. La définition des variables se fait à l'aide d'un mécanisme d'**axes de recherche**. Ce mécanisme est similaire à celui utilisé dans XPath[XPa99] pour localiser les parties d'un document.

Un motif sur une source possède les propriétés suivantes :
- un nom qui permet d'identifier le motif,
- une URL qui permet de spécifier la source de données à utiliser pour l'évaluation du motif,
- un axe de recherche qui permet de spécifier les données de la source à extraire en décrivant une forme à rechercher,
- éventuellement des conditions qui permettent de filtrer les données de la source.

La Figure 3.5 contient la partie de la DTD décrivant un motif sur une source.

```
<!ELEMENT source-pattern (search-axis, conditions?) >
<!ATTLIST source-pattern
          name   ID    #REQUIRED
          source IDREF #REQUIRED
>
```

FIG. 3.5 – Partie de la DTD décrivant un motif sur une source.

L'élément `source-pattern` est composé de deux sous-éléments `search-axis` et `conditions` qui représentent respectivement l'axe de recherche et les conditions du motif. L'attribut `name` est de type `ID` pour permettre d'identifier le motif sur la source. L'attribut `source` est de type `IDREF` pour référencer un élément indiquant la source de données du motif.

Source de données

La Figure 3.6 contient la partie de la DTD décrivant une source de données.

L'élément `source` est élément vide possédant deux attributs. L'attribut `id` de type `ID` permet d'identifier la source. Cet identifiant permet aux motifs de référencer une source. L'attribut `url` contient l'adresse de la source.

```
<!ELEMENT source EMPTY >
<!ATTLIST source
          id  ID    #REQUIRED
          url CDATA #REQUIRED >
```

FIG. 3.6 – Partie de la DTD décrivant une source de données.

Spécification d'un axe de recherche

Un axe de recherche contient un ensemble de nœuds qui décrivent la forme à rechercher dans la source (*source-node*). L'axe de recherche est évalué sur un **nœud contextuel** et possède une **fonction de recherche** qui permet de calculer les nœuds de la source qui devront être examinés. L'examen des nœuds de la source s'effectue en les comparant avec les nœuds décrivant la forme à rechercher (*source-nodes*).

La Figure 3.7 contient la partie de la DTD décrivant un axe de recherche.

```
<!ELEMENT search-axis (source-node+) >
<!ATTLIST search-axis
          function CDATA #REQUIRED
>
```

FIG. 3.7 – Partie de la DTD décrivant un axe de recherche.

L'élément `search-axis` est composé d'un ou plusieurs éléments `source-node` qui représentent les nœuds de la source à rechercher. L'attribut `function` contient la fonction utilisée pour rechercher les nœuds candidats de la source. Les fonctions de recherche proposées sont celles que nous avons définies dans le modèle de données : `children`, `children-comp`, `descendant`, `descendant-comp`. La sémantique de ces fonctions est expliquée dans la section consacrée aux opérations de navigation sur le graphe de données XML (3.2.3).

Spécification des nœuds de la source à rechercher

Pour décrire la forme des nœuds de données à rechercher dans la source, nous utilisons des nœuds appelés *source-node*. Ils sont composés :

- d'une expression régulière qui décrit le nom du nœud de la source,
- éventuellement d'un type qui spécifie le type du nœud à rechercher dans la source,
- éventuellement d'une variable qui sera instanciée pour les nœuds de la source respectant la spécification du motif (les variables déclarées dans un motif sur une source doivent avoir des noms distincts),
- éventuellement d'un axe de recherche qui permet de compléter la spécification du nœud à rechercher dans la source.

La Figure 3.8 contient la partie de la DTD décrivant un nœud de la source.

```
<!ELEMENT source-node (search-axis?) >
<!ATTLIST source-node
          reg-expression CDATA #REQUIRED
          type           CDATA #IMPLIED
          bindto         CDATA #IMPLIED
>
```

FIG. 3.8 – Partie de la DTD décrivant un nœud de la source.

L'attribut `reg-expression` contient l'expression régulière qui décrit le nom du nœud de la source à rechercher. L'attribut `type` contient éventuellement le type du nœud recherché. Conformément à notre modèle de données, les valeurs possibles pour cet attribut sont : `element`, `attribut` et `text`. L'attribut `binto` contient éventuellement une variable qui sera instanciée pour les nœuds de la source respectant la spécification de l'axe de recherche. Enfin, l'élément `source-node` possède éventuellement un sous-élément `search-axis` qui permet de spécifier la forme des fils ou des descendants du nœud. C'est l'imbrication des axes de recherche (`search-axis`) et des spécifications de nœuds sur les sources (`source-node`) qui permet de spécifier des structures complexes à rechercher dans une source de données.

Fonctionnement de la recherche des données à extraire

Nous allons illustrer maintenant le rôle des axes de recherche en décrivant leur fonctionnement. L'axe de recherche est évalué sur un **nœud contextuel**, appartenant à la source de données. Pour commencer l'évaluation, le nœud contextuel est l'élément racine du document contenant les données de la source. L'évaluation s'effectue en deux étapes.

1. La première étape évalue l'axe de recherche de la manière suivante.
   - La fonction de recherche est appliquée sur le nœud contextuel. Elle renvoie une liste de nœuds de la source qui seront candidats pour l'évaluation.
   - Chaque nœud de l'axe de recherche (*source-node*) évalue les nœuds candidats qui ont été renvoyés par la fonction de recherche. L'évaluation d'un nœud de l'axe de recherche nécessite un appel à la deuxième étape. Si tous les nœuds de l'axe de recherche peuvent être satisfaits par des nœuds candidats, alors l'axe de recherche renvoie toutes les instanciations possibles de ses variables[2].
2. L'évaluation d'un nœud de l'axe de recherche sur un nœud candidat se fait de la manière suivante.
   - On vérifie que le nom et le type du nœud candidat sont conformes à l'expression régulière et au type définis par le nœud de l'axe de recherche.
   - On évalue (s'il existe) l'axe de recherche du nœud. Le nœud contextuel de cette évaluation est le nœud candidat qui est en cours d'évaluation. Cette évaluation nécessite un appel à la première étape.

[2]On considère que les variables d'un axe de recherche sont celles contenues dans les nœuds de l'axe de recherche

On constate que les deux étapes de l'évaluation s'appellent récursivement (récursivité croisée). Ceci est dû au fait qu'un axe de recherche est composé de nœuds qui sont eux même composés éventuellement d'axes de recherche. La fin de la récursivité croisée est garantie par le fait que l'imbrication des axes de recherche et des nœuds de cet axe représente un arbre : un nœud ne peut pas contenir un axe de recherche qui a été défini à un niveau supérieur dans l'arbre.

Les algorithmes utilisés pour l'extraction des données des sources dans un SGBD relationnel sont présentés dans le chapitre 6 consacré à l'implémentation de notre prototype (§ 6.4.1, page 124).

Filtrage des données

Nous avons vu qu'un motif sur une source pouvait spécifier des conditions pour filtrer les données des sources à extraire. Pour cela, nous utilisons une composition de conditions qui s'appliquent aux variables définies dans le motif. La Figure 3.9 contient la partie de la DTD décrivant la composition de conditions.

```
<!ELEMENT conditions (condition | and | or) >
<!ELEMENT and ((condition | and | or), (condition | and | or)) >
<!ELEMENT or ((condition | and | or), (condition | and | or)) >

<!ELEMENT condition EMPTY >
<!ATTLIST condition
          left-expression  CDATA #REQUIRED
          operator         CDATA #REQUIRED
          right-expression CDATA #REQUIRED
>
```

FIG. 3.9 – Partie de la DTD décrivant la composition de conditions.

L'élément `conditions` spécifie une composition de conditions à l'aide des opérateurs logiques représentés par les éléments `and` et `or`. Chaque élément `condition` possède trois attributs indiquant la partie gauche, l'opérateur et la partie droite de la condition. L'expression des parties gauche et droite d'une condition permet d'utiliser des variables, des fonctions sur ces variables ou des constantes.

Exemple de motif sur une source

Considérons une source de données bibliographique validée par la DTD de la Figure 3.2. On veut construire un motif sur cette source pour extraire les noms et prénoms des auteurs. La Figure 3.10 contient le motif sur la source qui permet de spécifier les informations à extraire.

Le motif est identifié par l'attribut `name` dont la valeur est "`sp_auteurs_biblio`". La source de données est spécifiée par l'attribut `source` dont la valeur est "`biblio`", indiquant que la source de données est spécifiée dans un élément identifié par cette valeur.

```
<source-pattern name="sp_auteurs_biblio" source="biblio">
  <search-axis function="children">
    <source-node reg-expression="auteur" type="element">
      <search-axis function="children">
        <source-node reg-expression="nom"
                     type="element"
                     bindto="nom">
        </source-node>
        <source-node reg-expression="prenom"
                     type="element"
                     bindto="prenom">
        </source-node>
      </search-axis>
    </source-node>
  </search-axis>
</source-pattern>
```

FIG. 3.10 – Motif sur une source : noms et prénoms des auteurs.

Le premier élément `search-axis` est l'axe de recherche principal du motif sur la source. Le nœud contextuel de cet axe de recherche est l'élément racine de la source de données. Le `source-node` de cet axe spécifie la recherche des nœuds `auteur` qui sont fils de l'élément racine de la source. Ce nœud source possède lui aussi un axe de recherche qui spécifie deux nœuds de la source à rechercher. Ces deux éléments `source-node` permettent de rechercher des nœuds `nom` et `prenom` qui sont des sous-éléments de `auteur`. Ils sont liés à des variables pour utiliser les valeurs de ces éléments de la source.

3.3.2 Représentation logique des données

Les données extraites par un motif sur une source peuvent être **représentées d'un point de vue logique par une relation**[3]. L'instance de cette relation contiendra les données XML extraites de la source considérée. Chaque variable du motif est représentée par un attribut de la relation. Chaque instanciation possible du motif sur la source est représentée par un tuple dans l'instance de la relation.

Les données XML extraites de la source seront stockées dans les attributs de la relation représentant le motif sur la source. Pour cela, on peut utiliser un schéma générique, comme celui qui est présenté dans le chapitre 5 consacré à la matérialisation des vues (§ 5.2.3, page 85). Ce schéma permet de stocker des données XML dans une base de données relationnelle. Pour simplifier la présentation de notre modèle de vues, on considérera que les attributs d'une relation peuvent contenir des données XML.

Plus formellement, si on note $variables_{sp}$ l'ensemble des variables du motif sp, alors les données extraites de la source peuvent être représentées dans une relation R_{sp} de sorte $U_{R_{sp}} = variables_{sp}$ et dont l'instance sera définie par :

$$I(R_{sp}) = \{t \text{ définis sur } U_{R_{sp}} \mid t = instanciation(variables_{sp}, source_{sp})\}$$

[3]Au sens du modèle relationnel de CODD.

avec $instanciation(variables_{sp}, source_{sp})$ l'ensemble des instanciations possibles des variables du motif sp sur les données sa source, notée $source_{sp}$.

La conjonction de conditions éventuellement spécifiée par le motif peut être réalisée par l'opération de sélection de l'algèbre relationnelle, pour filtrer les données. Dans ce cas, les données extraites de la source sont :

$$\sigma_{conditions_{sp}}(I(R_{sp}))$$

avec $conditions_{sp}$ la conjonction de conditions permettant de filtrer les données extraites par le motif sp.

Exemple Considérons le motif sur la source `sp_auteurs_biblio` présenté dans la Figure 3.10. Ce motif définit deux variables pour extraire le nom et le prénom des auteurs. La sorte de la relation représentant ce motif sera donc composé de ces deux variables :

$$U_{sp\_auteurs\_biblio} = variables_{sp\_auteurs\_biblio} = \{nom, prenom\}$$

3.4 Union de données

Pour intégrer des sources de données multiples et hétérogènes, il est souvent nécessaire de définir l'**union** des données provenant de différentes sources. Pour cela, nous proposons l'utilisation de fragments : un fragment est une unité sémantique permettant d'intégrer des données provenant de sources multiples et hétérogènes.

Cette section est organisée de la manière suivante. Tout d'abord, nous présentons comment sont spécifiées les fragments avec VIMIX. Ensuite nous montrons comment les données spécifiées peuvent être représentées d'un point de vue logique par une table relationnelle.

3.4.1 Spécification d'un fragment

Un fragment est composé d'une liste de sources de données dont on veut faire l'union. Ces données peuvent provenir d'un motif sur une source, d'un autre fragment ou d'une jointure (que nous définirons dans la sous-section suivante). Nous avons vu que chaque motif permettait d'extraire des données d'une source, ces données pouvant être représentées par une relation. Les données intégrées dans un fragment (ou une jointure) peuvent aussi être représentées par une relation. Un fragment permet de spécifier l'union[4] des données extraites par chacune de ses sources.

De plus, un fragment permet de filtrer les données et de résoudre des conflits d'identité en éliminant les doublons dans les données provenant des différentes sources. Pour éliminer les doublons, nous

[4]L'opération d'union sera définie par la suite et nous la noterons $\bigoplus$.

proposons un mécanisme permettant de spécifier une source prioritaire. Les données intégrées dans le fragment, ne doivent pas être redondantes avec celles de la source prioritaire. Pour cela, on spécifie les variables qui permettent d'identifier les données de la source prioritaire.

Un fragment possède les propriétés suivantes :

- un nom qui permet d'identifier le fragment,
- une liste de sources (motif sur une source, fragment ou jointure) qui définissent les données dont on doit faire l'union (cette liste ne doit pas être vide),
- éventuellement des conditions qui permettent de filtrer les données du fragment,
- éventuellement un ensemble de restrictions qui permettent de résoudre des conflits d'identité de données.

La liste des sources d'un fragment f peut contenir des motifs, des fragments et des jointures. Les fragments et les jointures utilisés, ne doivent pas (eux-même ou les sources qu'ils utilisent) utiliser le fragment f. D'une manière plus générale, une source de données ne peut pas être définie en utilisant des données qui utilisent cette source. On peut donc construire, pour chaque source s, un arbre de sources permettant de définir les données de s.

La Figure 3.11 contient la partie de la DTD décrivant un fragment.

```
<!ELEMENT fragment (conditions?, restrictions*) >
<!ATTLIST fragment
          name ID     #REQUIRED
          data IDREFS #REQUIRED
>
```

FIG. 3.11 – Partie de la DTD décrivant un fragment.

L'attribut `name` de type `ID` permet d'identifier le fragment. Les données dont le fragment exprime l'union sont spécifiées par l'attribut `data` de type `IDREFS`. Les éléments référencés par l'attribut `data` doivent être des motifs sur les sources, des fragments ou des jointures (`source-pattern | fragment | join`)[5].

Le sous-élément `conditions` permet de spécifier des conditions pour filtrer les données dont le fragment fait l'union. Pour cela, nous utilisons une composition de conditions identique à celle utilisée pour filtrer les données d'un motif sur une source. La partie de la DTD décrivant l'élément `conditions` est présentée dans la Figure 3.9. De plus, l'élément `restriction` permet de résoudre des conflits d'identité entre les données provenant de plusieurs sources.

[5]Une DTD ne permet pas de spécifier ce type de contrainte car on ne peut pas spécifier le type des éléments cibles d'un attribut de type `IDREFS`.

Résolution de conflits

Les sources de données intégrées dans un fragment peuvent contenir des données conflictuelles. Nous proposons une solution pour résoudre les conflits d'identité de données au niveau du fragment. La propriété *resolution* contient éventuellement un ensemble de restrictions qui contiennent les propriétés suivantes :

- un ensemble de variables dont la valeur doit être unique (cet ensemble de variable doit être contenu dans toutes les sources utilisées par le fragment),
- la source de données (motif, fragment ou jointure) qui sera utilisée pour la résolution du conflit.

Une restriction permet donc de résoudre des conflits d'identité de données en indiquant quelle source est prioritaire.

La Figure 3.12 contient la partie de la DTD décrivant une restriction pour un fragment.

```
<!ELEMENT restriction EMPTY >
<!ATTLIST restriction
          variables CDATA #REQUIRED
          priority  IDREF #REQUIRED
>
```

FIG. 3.12 – Partie de la DTD décrivant une restriction.

L'attribut `variables` contient un ensemble de variable. Les tuples formés par cet ensemble de variables doivent être uniques dans les données extraites par le fragment. En cas de conflit, l'attribut `priority` indique la source à utiliser pour la résolution. Cet attribut est de type `IDREF` pour référencer une source de données. Cette source doit appartenir aux sources de données intégrées par le fragment. Plus formellement, si on note r_f une restriction du fragment f, alors $priority_{r_f} \in data_f$.

Exemple de fragment

Considérons une deuxième source de données bibliographiques, `biblio_lirmm` qui contient les publications et les auteurs du laboratoire. La DTD qui décrit cette source est présentée en annexe (B.2, page 156). Soit `sp_auteurs_lirmm` un motif qui permet d'extraire les auteurs de cette source, en fournissant le schéma suivant : $variables_{sp\_auteurs\_biblio\_lirmm} = \{nom, prenom, email\}$. La spécification de ce motif est donnée en annexe (C.2, page 159). On désire faire l'union des auteurs de cette source et de ceux de la source précédente (soit le motif `sp_auteurs_biblio` défini dans la Figure 3.10). La Figure 3.13 contient le fragment qui permet de spécifier l'union de ces données.

Le fragment est identifié par l'attribut `name` dont la valeur est "`f_auteurs`". L'attribut `data` dont la valeur est "`sp_auteurs_lirmm sp_auteurs`" permet de spécifier les sources dont on doit faire l'union. Les sources du fragment sont donc les motifs `sp_auteurs_biblio_lirmm` et `sp_auteurs_biblio`.

De plus, afin de ne pas introduire de redondance dans les données, on spécifie une restriction pour le fragment, avec l'élément `restriction`. L'attribut `variables` dont la valeur est `nom` spécifie qu'il

```
<fragment name="f_auteurs" data="sp_auteurs_biblio_lirmm sp_auteurs_biblio">
  <restrictions>
    <restriction variable="nom" priority="sp_auteurs_biblio_lirmm" />
  </restrictions>
</fragment>
```

FIG. 3.13 – Fragment spécifiant l'union des auteurs.

n'y aura pas de tuples ayant des valeurs de `nom` redondants pour ce fragment. L'attribut `priority` dont la valeur est "`sp_auteurs_biblio_lirmm`" spécifie la la source à utiliser pour résoudre ce conflit.

3.4.2 Représentation logique des données d'un fragment

Les données extraites par un fragment peuvent être représentées d'un point de vue logique par une relation. La sorte de cette relation est définie par l'union de l'ensemble des variables des sources utilisées par le fragment :

$$U_{R_f} = \bigcup_{d \in data_f} variables_d$$

avec $data_f$ la liste des sources de données de f et $variables_d$ l'ensemble des variables d'une source de données d. Cette source d peut être un motif, un fragment ou une jointure.

L'instance de cette relation est définie par l'union des données provenant des sources du fragment. L'opération d'union dans l'algèbre relationnelle est définie pour des relations de même sorte, c'est à dire ayant le même ensemble d'attributs. Comme les sources qui composent un fragment peuvent avoir des sortes différentes (c'est à dire qu'elles définissent des variables différentes), nous avons défini une opération d'union, notée $\bigoplus$ qui s'applique aussi à des relations ayant des sortes différentes. Cette opération construit une relation ayant pour sorte l'union des attributs auxquels elle s'applique. Elle fonctionne comme l'union définie dans l'algèbre relationnelle pour les attributs qui appartiennent aux deux relations et complète les tuples avec la valeur `NULL` pour les attributs qui ne sont pas définis dans la relation d'origine. On peut décrire plus formellement cette opération avec la définition suivante (où $setofR$ est un ensemble de relations) :

$$\begin{aligned} I(\bigoplus setofR) = \quad \{ \quad & t \text{ définis sur } U_{setofR} = \textstyle\bigcup_{R \in setofR} U_R \mid \\ & \forall R \in setofR, \forall s \in I(R) : \\ & \begin{cases} \wedge & \pi_{U_R}(t) = s \\ & \pi_x(t) = (NULL), \forall x \in U_{setofR} - U_R \end{cases} \\ \} & \end{aligned}$$

L'expression suivante illustre le fonctionnement de cette opération par un exemple simple (les relations sont représentées sous forme tabulaire) :

a	b
x	y

$\oplus$

a	b	c
x	y	z

$=$

a	b	c
x	y	NULL
x	y	z

L'instance de la relation représentant les données d'un fragment est définie en utilisant ce nouvel opérateur :

$$I(R_f) = \bigoplus_{d \in data_f} (R_d)$$

avec $data_f$ qui note l'ensemble des relations représentant les sources du fragment.

Les conditions éventuellement définies sur le fragment peuvent être utilisées par l'opération de sélection de l'algèbre relationnelle pour filtrer les données. Dans ce cas, les données extraites par le fragment sont :

$$\sigma_{conditions}(I(R_f))$$

Les restrictions éventuellement définies sur le fragment peuvent s'exprimer à l'aide de l'algèbre relationnelle. Les conflits sont résolus avant d'effectuer l'union (avec $\bigoplus$) sur les relations représentant les sources du fragment. Si on note A l'ensemble des attributs de la restriction et D le relation représentant la source qui permet sa résolution, alors on doit effectuer :

$$\forall R \in data(f) - D : (\pi_A(R) - \pi_A(D)) \bowtie R$$

En d'autres termes, sur chacune des relations qui représentent les motifs du fragment (à l'exception de celle permettant la résolution) on supprime les tuples qui possèdent des attributs redondants avec ceux de la source prioritaire.

Exemple Considérons le fragment `f_auteurs` présenté dans la Figure 3.13. Ce fragment définit l'union de deux motifs sur les sources, qui peuvent être représentés par les relations suivantes :

$$\begin{array}{ll} sp\_auteurs\_biblio\_lirmm & (nom, prenom, email) \\ sp\_auteurs\_biblio & (nom, prenom) \end{array}$$

Le schéma de la relation représentant le fragment sera donc :

$$f\_auteurs(nom, prenom, email)$$

3.5 Jointure de données

Pour intégrer des sources de données multiples et hétérogènes, il est souvent nécessaire de définir la jointure de données provenant de différentes sources. Nous avons utilisé une opération de jointure avec prédicat qui permet d'effectuer des jointures entre les données définies par différentes sources.

Cette section est organisée de la manière suivante. Tout d'abord, nous présentons comment sont spécifiées les jointures avec VIMIX. Ensuite nous montrons comment les données spécifiées peuvent être représentées d'un point de vue logique par une table relationnelle.

3.5.1 Spécification d'une jointure

Une jointure est constituée d'un nom qui permet de l'identifier, d'une partie gauche et d'une partie droite. Chaque partie comprend une source de données (qui peut être un motif, un fragment ou une autre jointure) et une variable qui est définie dans cette source. Une jointure possède donc les propriétés suivantes :

Comme nous l'avons vu dans la section précédente consacrée à la spécification des fragments, une source s ne pouvait pas utiliser des sources utilisant s pour dans sa définition. Pour les mêmes raisons, une jointure j ne peut pas utiliser des sources utilisant j dans leur définition.

La Figure 3.14 contient la partie de la DTD décrivant une jointure.

```
<!ELEMENT join EMPTY >
<!ATTLIST join
          name           ID    #REQUIRED
          left-data      IDREF #REQUIRED
          left-variable  CDATA #REQUIRED
          right-data     IDREF #REQUIRED
          right-variable CDATA #REQUIRED
>
```

FIG. 3.14 – Partie de la DTD décrivant une jointure.

L'attribut `name` de type `ID` permet d'identifier la jointure. Les sources de données des parties gauche et droite de la jointure sont spécifiées respectivement par les attributs `left-source` et `right-source`. Les éléments référencés par ces attributs doivent être des motifs sur les sources, des fragments ou des jointures (`source-pattern | fragment | join`)[6]. Les variables utilisées pour spécifier le prédicat de jointure sont spécifiées par les attributs `left-variable` et `right-variable`, pour spécifier respectivement la variable de la source référencée par `left-source` et celle référencée par `right-source`.

Exemple de jointure

Considérons une troisième source de données bibliographiques, `librairie` qui contient les livres d'une librairie. La DTD qui décrit cette source est présentée en annexe ((B.3, page 157)). Soit `sp_livres` un motif qui permet d'extraire les livres avec les noms de leurs auteurs, en fournissant le schéma suivant : $variables_{sp\_livres} = \{titre, auteur, prix\}$. La spécification de ce motif est donnée en annexe (C.3, page 160). On désire "croiser" les données de cette source avec les auteurs, afin d'obtenir les livres écrits par des auteurs du LIRMM. Pour cela, on doit faire la jointure entre le motif sur la source de la librairie et le fragment qui contient tous les auteurs. La Figure 3.15 contient la jointure qui permet d'exprimer le croisement de ces informations.

[6]Une DTD ne permet pas de spécifier ce type de contrainte car on ne peut pas spécifier le type de l'élément cible d'un attribut de type `IDREF`.

```
<join name="j_livres_lirmm"
      left-data="sp_livres"
      left-variable="auteur"
      right-data="f_auteurs"
      right-variable="nom"
/>
```

FIG. 3.15 – Jointure des livres et des auteurs du LIRMM.

3.5.2 Représentation logique des données d'une jointure

Les données intégrées dans une jointure peuvent être représentées d'un point de vue logique par une relation. La sorte de cette relation est définie par l'union de l'ensemble des variables des sources utilisées par la jointure :

$$U(R_j) = variables_{ld_j} \cup variables_{rd_j}$$

avec $variables_{ld_j}$ et $variables_{rd_j}$ respectivement l'ensemble des variables de la source de données (motif, fragment ou jointure) de gauche et de droite de la jointure j. De plus, on doit renommer les attributs provenant des sources de données de gauche et de droite[7] afin d'éviter les conflits de noms entre les attributs de la relation représentant la jointure.

L'algèbre relationnelle dispose d'une opération de jointure naturelle qui utilise comme prédicat les attributs communs aux deux relations dont on veut calculer la jointure. Nous utiliserons ici l'opération de **jointure avec prédicat** pour laquelle le prédicat de jointure devra être spécifié explicitement. En effet, certaines sources peuvent avoir des attributs de même nom qui ne doivent pas constituer le prédicat de jointure. L'instance de la relation R_j est définie en utilisant la jointure avec prédicat des données des sources :

$$I(R_j) = R_{ld_j} \bowtie_{lv_j = rv_j} R_{rd_j}$$

avec ld_j, rd_j, lv_j et rd_j les sources de données et les variables des parties de gauche et de droite de la jointure j.

Exemple Considérons la jointure `j_livres_lirmm` présentée dans la Figure 3.15. La partie gauche de cette jointure est un motif dont le schéma est $variables_{sp\_livres} = \{titre, auteur, prix\}$, tandis que la partie droite est le fragment dont le schéma est $variables_{f\_auteurs} = \{nom, prenom, email\}$. Le schéma de la relation représentant la jointure sera donc :

$$f\_auteurs \quad (\quad sp\_livres\_titre, sp\_livres\_auteur, sp\_livres\_prix, \\ f\_auteurs\_nom, f\_auteurs\_prenom, f\_auteurs\_email \quad)$$

[7] Par exemple, on peut renommer les attributs des sources en les préfixant par le nom de la source. On garantit ainsi l'unicité de chaque nom d'attribut de la relation représentant la jointure

3.6 Spécification d'une vue

Une vue peut être définie par un n-uplet qui possède les propriétés suivantes :

- un nom qui permet d'identifier la vue,
- une source (motif, fragment ou jointure) qui contient les données de la vue,
- un motif qui décrit la structure du résultat de la vue,
- l'ordre des données,
- les niveaux de regroupement des données.

La Figure 3.16 contient la partie de la DTD décrivant une vue.

```
<!ELEMENT view (result-node) >
<!ATTLIST view
          name     ID    #REQUIRED
          data     IDREF #REQUIRED
          order-by CDATA #IMPLIED
          group-by CDATA #IMPLIED
>
```

FIG. 3.16 – Partie de la DTD décrivant une vue.

L'attribut `name` de type `ID` permet d'identifier la vue. Les données de la vue sont spécifiées par l'attribut `source` de type `IDREF`. L'élément référencé par l'attribut `source` doit être un motif sur une source, un fragment ou une jointure (`source-pattern | fragment | join`). Ces éléments ont été présentés dans le chapitre 3 consacré à la spécification des données à extraire.

Le sous-élément `result-node` permet de spécifier la forme du résultat de la vue. Il permet de représenter un arbre qui spécifie la structure et les données de la vue.

L'attribut `order-by` permet de spécifier l'ordre des données dans le résultat de la vue. Cet attribut contient une liste de variables. Toutes ces variables doivent être définies dans la source de données utilisée.

Enfin, l'attribut `order-by` permet de spécifier des niveaux de regroupement des données. Cet attribut contient une liste de variables. Toutes ces variables doivent être définies dans la source de données utilisée.

3.6.1 Structure du résultat d'une vue

Arbre décrivant la forme du résultat

La forme du résultat d'une vue est définie par un arbre qui décrit sa structure. Cet arbre permet de spécifier (i) la forme du document qui représentera la vue et (ii) les données de la source utilisée pour peupler le résultat de la vue. Pour cela, il est composé de trois types de nœuds :

- le type `element` permet de spécifier un élément dans le résultat de la vue,
- le type `attribute` permet de spécifier un attribut dans le résultat de la vue,
- le type `expression` permet de spécifier les données à insérer dans le résultat de la vue.

La Figure 3.17 contient la partie de la DTD décrivant l'arbre spécifiant le résultat d'une vue.

```
<!ELEMENT result-node (result-node*) >
<!ATTLIST result-node
          type  CDATA #REQUIRED
          value CDATA #REQUIRED
>
```

FIG. 3.17 – Partie de la DTD décrivant la forme du résultat d'une vue.

L'élément `result-node` permet de définir un arbre. Pour cela, il est composé de zéro ou plusieurs fils, permettant ainsi de définir une structure hiérarchique. La racine de l'arbre est le sous-élément `result-node` qui est le fils de l'élément `view`.

L'attribut `type` indique le type du nœud de l'arbre. Il peut prendre les valeurs "`element | attribute | expression`". Certaines contraintes ne sont pas exprimées par la DTD.

- Le nœud représentant la racine de l'arbre doit être obligatoirement de type `element`.
- Un nœud de type `attribute` ne doit pas avoir de fils de type `element` ou `attribute`, car cette construction est interdite dans le langage XML.
- Les nœuds de type `expression` sont obligatoirement des feuilles de l'arbre.

Enfin, l'attribut `value` contient la valeur du nœud. La sémantique de cette valeur dépend du type du nœud.

- Pour un nœud de type `element`, la valeur indique le nom de l'élément qui sera construit pour peupler le résultat de la vue.
- Pour un nœud de type `attribute`, la valeur indique le nom de l'attribut qui sera construit pour peupler le résultat de la vue.
- Pour un nœud de type `expression`, la valeur permet d'exprimer une expression décrivant les données à insérer pour peupler le résultat de la vue.

Expression spécifiant les données de la vue

Nous avons vu dans le chapitre précédent, consacré à la spécification des données à extraire, que ces données pouvaient être représentées d'un point de vue logique par des tables relationnelles. Les données "atomiques" de ces tables relationnelles sont des nœuds définis selon notre modèle de données pour XML. La spécification d'une vue doit permettre de restructurer les données. Pour cela, nous proposons des fonctions qui s'appliquent aux nœuds qui ont été extraits des sources XML.

Les opérations définies sur les nœuds du graphe de données XML ((§ 3.2.3, page 42)) peuvent être utilisées dans les nœuds de type `expression` de l'arbre spécifiant le résultat de la vue. Les fonctions les plus utiles sont `text()` et `r-text()` qui permettent de spécifier la représentation textuelle d'un nœud élément ou attribut.

La spécification du langage XML [XML00] ne propose pas de type numérique[8]. De plus, dans les sources de données, les valeurs numériques peuvent être mélangées avec du texte. Nous proposons deux fonctions qui permettent d'extraire des valeurs numériques dans du texte. Ces fonctions permettent d'extraire des nombres entiers et des nombres réels dans du texte. Nous les avons définies pour que leur utilisation et leur implémentation soient aisées. Cependant, il existe des méthodes plus perfectionnées pour extraire des valeurs numériques (et plus généralement de l'information) dans du texte. Notamment, EXREP [LQVC95] est un outil générique de réécriture basé sur les expressions régulières. Il permet d'extraire des informations dans du texte en définissant des filtres.

La fonction `int(string, regular-expression)` → `int` permet d'extraire un nombre entier dans une chaîne de caractères. Les deux paramètres de cette fonction sont la chaîne de caractère à traiter et une expression régulière qui spécifie la forme à rechercher pour extraire la valeur numérique. Cette expression régulière doit contenir un caractère "`%`" qui désigne le nombre entier à extraire. Le caractère "`*`" permet de spécifier n'importe quelle chaîne de caractères. Enfin, le caractère "`|`" permet de spécifier un choix entre plusieurs chaînes de caractères. Nous présentons ici quelques exemples d'utilisation de cette fonction.

- `int("40", "%")` → 40,
- `int("40 Euros", "% Euros")` → 40,
- `int("40 Eur", "% Eur")` → NULL,
- `int("40 Euros", "% (Eur|Euros)")` → 40,
- `int("40 Euros", "% *")` → 40,
- `int("vaut 40 Eur", "% *")` → NULL,
- `int("vaut 40 Euros", "* % *")` → 40.

La fonction `float(string, regular-expression)` → `float` permet d'extraire un nombre réél dans une chaîne de caractères. Les deux paramètres de cette fonction sont la chaîne de caractère à traiter et une expression régulière qui spécifie la forme à rechercher pour extraire la valeur numérique. Cette expression régulière doit contenir deux caractères "`%`", le premier désignant la partie entière du nombre réel et le second sa partie décimale. Nous présentons ici quelques exemples d'utilisation de cette fonction.

- `int("39.99 Euros", "%.% Euros")` → 39.99,
- `int("39,99 Euros", "%.% Euros")` → NULL,
- `int("39,99 Euros", "%*% Euros")` → 39.99,
- `int("39 Euros 99 centimes", "% Euros % centimes")` → 39.99,
- `int("39 Euros et 99 centimes", "%*%*")` → 39.99,

On pourra aussi utiliser une notation abrégée, qui permet de simplifier l'écriture des expressions :

- `int(node, regular-expression)` ⟺ `int(text(node), regular-expression)`,

[8] Cependant, XML S*chema* [XML01a] permet de spécifier que des valeurs doivent être de type numérique

- `int(node)` $\Longleftrightarrow$ `int(text(node), "*%*")`,
- `float(node, regular-expression)` $\Longleftrightarrow$ `float(text(node), regular-expression)`,
- `int(node)` $\Longleftrightarrow$ `int(text(node), "*%*%*")`.

Enfin, les expressions peuvent contenir des fonctions d'agrégation lorsque des niveaux de regroupement ont été définis dans la vue. Ces fonctions sont présentées dans la suite.

Exemple de vue

Considérons le fragment `f_auteurs` (Figure 3.13, page 53) qui spécifie l'union de deux sources de données bibliographiques pour extraire les auteurs. Nous allons spécifier une vue qui présente les noms et prénoms des auteurs de manière uniforme. Le résultat de la vue sera structuré de la manière suivante : chaque élément `auteur` possédera deux attributs contenant le nom et le prénom. La figure Figure 3.18 présente la spécification de cette vue.

```
<view name="v_auteurs" source="f_auteurs" group-by="" order-by="">
  <result-node type="element" value="auteur">
    <result-node type="attribute" value="nom">
      <result-node type="expression" value="text(nom)" />
    </result-node>
    <result-node type="attribute" value="prenom">
      <result-node type="expression" value="text(prenom)" />
    </result-node>
  </result-node>
</view>
```

FIG. 3.18 – Vue présentant de manière uniforme les noms et prénoms des auteurs.

L'élément `view` contient la spécification de la vue. L'attribut `name` dont la valeur est "`v_auteurs`" permet d'identifier cette vue. L'attribut `source` dont la valeur est "`f_auteurs`" permet de spécifier la source de données à utiliser.

L'arbre spécifiant le résultat de la vue est représenté par l'élément `source-node`. La racine spécifie un élément `auteur`. Elle possède deux sous-éléments de type attribut. Chacun de ces attributs ont un fils de type `expression` qui spécifie leur valeur. L'expression de l'attribut `nom` dont la valeur est `text(nom)` exprime que le nœud qui peuplera le résultat de la vue sera la représentation textuelle de la variable `nom`. De même, l'expression de l'attribut `prenom` exprime que la valeur de cet attribut sera la représentation textuelle de la variable `prenom`.

3.6.2 Niveaux de regroupement et fonctions d'agrégation

Niveaux de regroupement

Notre langage de définition de vues permet de définir également des niveaux de regroupement sur les variables de la source de données utilisée.

Le fonctionnement de ces niveaux de regroupement est basé sur les possibilités offertes par le langage SQL pour créer des partitions de données à l'aide de la clause `group by`. La table représentant les données de la vue constitue la partition initiale. Lorsqu'on définit un niveau de regroupement sur une variable de cette partition, elle est découpée en autant de partitions qu'il existe de valeurs différentes pour la variable choisie. Le schéma de ces nouvelles partitions est composé des variables de la partition initiale à l'exception de la variable utilisée pour définir le niveau de regroupement. Lorsqu'il existe plusieurs niveaux de regroupement, on répète ce processus en considérant tour à tour chacune des partitions générées comme la partition initiale.

De plus, les niveaux de regroupement permettent d'utiliser des fonctions d'agrégation pour intégrer les données des sources.

Fonctions d'agrégation

Les partitions générées par la spécification de niveaux de regroupement permettent d'appliquer des fonctions d'agrégation aux variables. Lorsque ces fonctions acceptent en paramètre un ensemble de valeurs numériques, on doit utiliser les fonctions `int()` ou `float()` pour transformer les nœuds de données XML liés aux variables de la vue. Les nœuds d'une partition ainsi transformés pourront être utilisés en paramètres des fonctions d'agrégation.

Nous proposons les fonctions d'agrégations classiques présentes dans le langage SQL.

- La fonction `count()` permet de compter les lignes d'une partition. C'est la seule fonction d'agrégation qui ne nécessite pas de paramètre.
- La fonction `sum()` renvoie la somme de l'ensemble des valeurs numériques passées en paramètre.
- La fonction `avg()` renvoie la moyenne de l'ensemble des valeurs numériques passées en paramètre.
- La fonction `max()` renvoie la valeur maximale de l'ensemble des valeurs numériques passées en paramètre.
- La fonction `min()` renvoie la valeur minimale de l'ensemble des valeurs numériques passées en paramètre.

Exemple de vue

Considérons la jointure `j_livres_lirmm` (Figure 3.15, page 56) qui spécifie le croisement des livres et des auteurs de deux sources de données bibliographiques. Nous allons spécifier une vue qui présente pour chaque auteur : son nom, le nombre de livres qu'il a écrit, le prix moyen des livres qu'il a écrit et les titres de ses livres. Le résultat de la vue sera structuré de la manière suivante : chaque élément `auteur`

possédera trois attributs contenant le nom, le nombre de livre et leur prix moyen. Les titres des livres d'un auteur seront contenus dans des sous-éléments. La figure Figure 3.19 présente la spécification de cette vue.

```
<view name="v_livres_lirmm"
      source="j_livres_lirmm"
      order-by="auteur"
      group-by="auteur">
  <result-node type="element" value="auteur">
    <result-node type="attribute" value="nom">
      <result-node type="expression" value="text(auteur)" />
    </result-node>
    <result-node type="attribute" value="nb-livres">
      <result-node type="expression" value="count()" />
    </result-node>
    <result-node type="attribute" value="prix-moyen">
      <result-node type="expression" value="avg(float(prix))" />
    </result-node>
    <result-node type="element" value="livre">
      <result-node type="expression" value="text(titre)" />
    </result-node>
  </result-node>
</view>
```

FIG. 3.19 – Vue intégrant des informations sur les livres de chaque auteur.

L'élément `view` contient la spécification de la vue. L'attribut `name` dont la valeur est "`v_livres_lirmm`" permet d'identifier cette vue. L'attribut `source` dont la valeur est "`j_livres_lirmm`" permet de spécifier la source de données à utiliser.

L'arbre spécifiant le résultat de la vue est représenté par l'élément `source-node`. La racine spécifie un élément `auteur`. Elle possède quatre sous-éléments représentant les trois attributs et le sous-élément `livre`.

L'expression de l'attribut `nom` dont la valeur est `text(auteur)` exprime que le nœud qui peuplera le résultat de la vue sera la représentation textuelle de la variable `auteur`.

L'expression de l'attribut `nb-livres` dont la valeur est `count()` exprime que le nœud qui peuplera le résultat de la vue sera calculé avec la fonction d'agrégation `count`. Le résultat de cette fonction comptera tous les livres de l'auteur.

L'expression de l'attribut `prix-moyen` dont la valeur est `avg(float(prix))` exprime que le nœud qui peuplera le résultat de la vue sera calculé avec la fonction d'agrégation `avg`. Le résultat de cette fonction sera le prix moyen des livres de l'auteur.

L'élément `livre` possède un fils expression qui spécifie qu'il sera peuplé par la valeur textuelle de la variable `titre`. Du fait du niveau de regroupement défini par la vue (sur la variable `auteur`), l'élément `livre` sera répété autant de fois qu'il y a de titres différents pour un auteur.

Chacun de ces attributs a un fils de type `expression` qui spécifie sa valeur. De même, l'expression de l'attribut `prenom` exprime que la valeur de cet attribut sera la représentation textuelle de la variable `prenom`.

3.6.3 Génération du schéma de la vue

La structure spécifiée par le résultat de la vue permet de générer un schéma (sous forme de DTD) qui validera le document représentant le résultat de la vue.

Pour générer une DTD qui validera le document représentant le résultat de la vue, on utilise l'arbre qui spécifie la structure du résultat et les niveaux de regroupement indiqués par la clause `group-by`.

La génération de la DTD s'effectue en appliquant les règles de transformation suivantes :

1. Le nœud racine de la DTD est un élément dont le nom est celui de la vue.
2. L'élément racine de la DTD est composé de zéro ou plusieurs éléments représentant le nœud racine de l'arbre représentant la structure du résultat.
3. Chaque élément de la DTD représentant un nœud élément de la spécification possède :
   - les éléments représentant les nœuds fils de type élément,
   - une liste d'attributs représentant les nœuds fils de type attribut,
   - `#PCDATA` pour les nœuds fils de type expression.

   Cette règle s'applique récursivement.
4. Les niveaux de regroupement indiquent que pour les variables utilisées on créé une partition. Lorsque qu'on a un niveau de regroupement, pour chaque valeur de la variable utilisée, on associe toutes les valeurs de la partition. La variable utilisée pour définir le niveau de regroupement apparaît dans un nœud de type expression. Tous les nœuds frères des ancêtres (à un niveau quelconque) de ce nœud peuvent apparaître plusieurs fois. Pour exprimer cela, on ajoute le symbole `+` dans la DTD. Cette règle s'applique pour chaque niveau de regroupement.

DTD de la vue `v_auteurs`

La Figure 3.20 présente le schéma généré pour la vue `v-auteurs` (décrite dans la Figure 3.18).

```
<!ELEMENT v_auteurs (auteur*) >
<!ELEMENT auteur EMPTY >
<!ATTLIST auteur
          nom CDATA #REQUIRED
          prenom CDATA #REQUIRED >
```

FIG. 3.20 – Schéma généré pour la vue `v_auteurs`.

Cette DTD a été construite en appliquant les règles présentées plus haut. L'élément racine de la DTD est un élément `v_auteurs` (règle 1). Il est composé de zéro ou plusieurs éléments `auteur` (règle 2). L'élément `auteur` possède deux attributs `nom` et `prenom` (règle 3).

DTD **de la vue** `v_livres_lirmm`

La Figure 3.21 présente le schéma généré pour la vue `v-livres-lirmm` (décrite dans la Figure 3.19).

```
<!ELEMENT v_livres_lirmm (auteur*) >
<!ELEMENT auteur (livre+) >
<!ATTLIST auteur
          nom CDATA #REQUIRED
          nb-livres CDATA #REQUIRED
          prix-moyen CDATA #REQUIRED >
<!ELEMENT livre (#PCDATA) >
```

FIG. 3.21 – Schéma généré pour la vue `v_livres_lirmm`.

Cette DTD a été construite en appliquant les règles présentées plus haut. L'élément racine de la DTD est un élément `v_livres_lirmm` (règle 1). Il est composé de zéro ou plusieurs éléments `auteur` (règle 2). L'élément `auteur` possède trois attributs `nom`, `nb-livres`, `prix-moyen` et un sous élément `livre` (règle 3). L'élément `livre` est de type `#PCDATA` car il possède un fils de type `expression` (règle 3). Le niveau de regroupement défini sur la variable `auteur` implique l'ajout du caractère "`+`" pour spécifier que le sous-élément `livre` peut apparaître plusieurs fois (règle 4).

3.7 Conclusion

Notre modèle de vues, nommé VIMIX, est composé d'un modèle de données pour XML et d'un langage qui permet de définir des vues. La syntaxe de ce langage de définition de vues utilise XML. Nous avons présenté la partie du langage qui permet de spécifier les données des sources à extraire.

Nous utilisons un **graphe de données** XML pour représenter les données d'un document XML. Nous avons pris en compte les liens de composition et les liens de référence entre les éléments. Les liens de composition représentent l'imbrication des éléments dans les données. Les liens de référence représentent les liens qui sont définis grâce au mécanisme d'attributs `ID/IDREF(S)` pour le partage des éléments.

Lorsqu'on considère uniquement les liens de composition entre les éléments, le sous-graphe représentant les données est un arbre. Cette propriété est intéressante, car elle permet d'utiliser des algorithmes plus efficaces pour traiter les données lorsque la sémantique des liens de référence n'est pas nécessaire. De plus, nous proposons des fonctions qui permettent de naviguer dans les données du graphe.

Notre **langage de définition de vues** permet de spécifier et de restructurer les données des sources à extraire grâce aux concepts suivants.

- Les **motifs** sur les sources permettent de spécifier les données à extraire, en utilisant le *pattern-matching*. D'un point de vue logique, les données extraites peuvent être représentées par une table relationnelle.
- Les **fragments** permettent d'intégrer des données provenant de plusieurs sources, en redéfinissant l'opération d'union de l'algèbre relationnelle.
- Les **jointures** permettent de "croiser" des données de deux sources, en utilisant la jointure de l'algèbre relationnelle avec un prédicat.

La spécification du résultat décrit la structure et les données de la vue. Elle utilise une source de données pouvant être un motif, un fragment ou une jointure définie précédemment. Pour cela, le résultat de la vue est défini par un **arbre** contenant (1) des nœuds de type `element` et `attribute` spécifiant la structure des données et (2) des nœuds de type `expression` qui spécifient les données. Les expressions utilisées permettent de construire la représentation textuelle des données des sources. L'utilisation de fonctions d'agrégation est également possible. Pour cela, nous avons défini des fonctions qui permettent d'extraire des valeurs numériques dans des nœuds de données XML.

Notre langage offre de riches possibilités pour la restructuration des données : création de **nouveaux éléments**, utilisation de **niveaux de regroupement**, de **fonctions d'agrégation**. De plus, la structure de la spécification d'une vue permet de **générer un schéma** sous la forme d'une DTD qui valide les données du résultat.

Un mécanisme de définition de vues offre de nombreuses applications dans le contexte des SGBD. En effet, il permet de présenter des données sous différents "points de vues" à des utilisateurs ou des groupes d'utilisateurs. On peut donc l'utiliser pour personnaliser la présentation des données, assurer la confidentialité des données, ...

De plus, comme nous le présentons dans le chapitre suivant, VIMIX permet de définir des vues pour intégrer des données. Nous présentons également un mécanisme d'aide qui facilite la construction de motifs sur les sources.

Chapitre 4

Spécification de vues VIMIX pour l'intégration

4.1 Introduction

Dans ce chapitre, nous présentons l'utilisation de notre modèle de vues pour intégrer des données provenant de sources XML hétérogènes. Notre langage de définition de vues permet de résoudre certains conflits entre les données des sources. Le choix d'XML permet de résoudre les problèmes syntaxiques de l'intégration. Cependant, il faut résoudre les problèmes liés à l'hétérogénéité de la structure des différentes sources à intégrer. De plus, il peut y avoir des conflits de données entre les différentes sources (données redondantes). Pour intégrer des données, VIMIX offre les possibilités suivantes :

- Faire l'union de plusieurs sources (`fragment`).
- Faire la jointure de plusieurs sources (`join`).
- Eliminer les redondances provenant de sources multiples (`restriction`).

De plus, la spécification d'une vue permet de générer le schéma de son résultat. Dans le processus d'intégration de données, le schéma médiateur est défini comme un ensemble de vues. Les schémas générés pour les vues permettent de construire le schéma médiateur d'une collection de vues.

Lors de l'intégration de sources hétérogènes, la structure des données n'est pas toujours parfaitement connue par le concepteur des vues. Il est alors difficile de construire un motif sur une source spécifiant les données à extraire, car le motif décrit la structure de la source. Pour faciliter cette tâche, nous avons proposé un mécanisme d'aide. Ce mécanisme propose une liste de choix possibles lors de la spécification des nœuds d'un motif sur une source.

Ce chapitre est organisé comme suit.

- Dans la section 4.2, nous présentons l'utilisation de VIMIX pour construire un schéma médiateur intégrant des données XML provenant de sources multiples et hétérogènes. Notre approche peut être qualifiée de mixte, car elle présente certains avantages des approches GAV et LAV.

– Dans la section 4.3, nous présentons les mécanismes d'aide que nous avons proposés pour permettre au concepteur d'un système d'intégration de faciliter la spécification des motifs sur les sources. Pour cela, nous utilisons la DTD de la source si elle existe ou un *dataguide* qui est un résumé de la structure d'une source de données XML.

4.2 Intégration de sources de données XML

La Figure 4.1 résume le mécanisme d'intégration de données proposé par notre approche. Il se décompose en trois étapes.

1. Spécification des données à extraire : lors de cette étape, on définit des motifs sur les sources (*source-pattern*). Ces motifs constituent les feuilles du graphe de *mappings*.
2. Union et jointure des données : lors de cette étape, on définit des fragments et des jointures qui permettent de construire le graphe de *mappings*.
3. Spécification de vues : lors de cette étape, on restructure les données extraites à partir des nœuds du graphe de *mappings*.

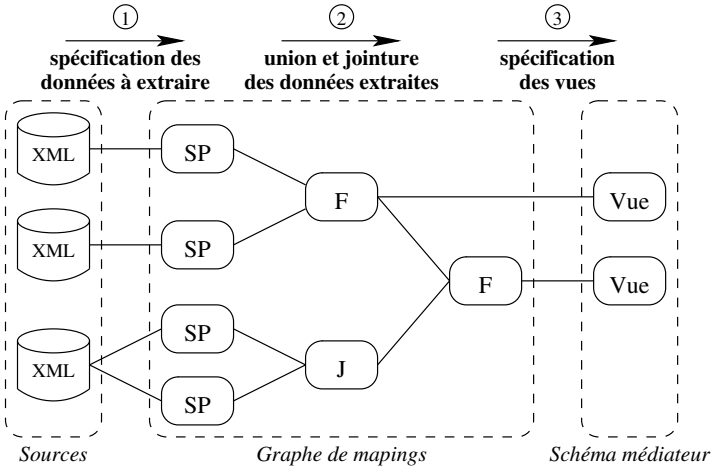

FIG. 4.1 – Intégration de données avec VIMIX.

Nous présentons ici comment notre modèle VIMIX permet de définir un schéma médiateur à partir d'un ensemble de vues.

4.2.1 Spécification du schéma médiateur

La Figure 4.2 contient la partie de la DTD décrivant un schéma médiateur composé de vues VIMIX.

L'élément `mediated-schema` est composé de trois sous-éléments qui permettent de définir le schéma médiateur pour intégrer des données XML.

L'élément `sources` contient les sources de données à intégrer. Il est composé d'un ou plusieurs éléments `source`, présenté dans la Figure 3.6.

```
<!ELEMENT mediated-schema (sources, extracted-data, views) >
<!ELEMENT sources (source+) >
<!ELEMENT extracted-data (source-pattern+, (fragment | join)*) >
<!ELEMENT views (view+) >
```

FIG. 4.2 – Partie de la DTD décrivant la spécification d'un système d'intégration.

L'élément `extracted-data` contient la spécification des données des sources à extraire. Il est composé d'un ou plusieurs éléments `source-pattern` (présenté dans la Figure 3.5) et de zéro ou plusieurs éléments `fragment` ou `join` (présentés respectivement dans les Figures 3.11 et 3.14).

Enfin, l'élément `views` contient les vues VIMIX qui définissent le schéma médiateur. Il est composé d'un ou plusieurs éléments `view`, présenté dans la Figure 3.16. Ces vues permettent d'intégrer les données des sources. Pour cela, elles sont définies en utilisant la spécification des données à extraire.

Exemple de schéma médiateur

Nous allons illustrer la spécification d'un schéma médiateur avec notre langage de définition de vues. Pour cela, nous allons spécifier un système intégrant des données bibliographiques en utilisant les vues `v_auteurs` (Figure 3.18) et `v_livres_lirmm` (Figure 3.19).

La Figure 4.3 présente la spécification de ce schéma médiateur.

```
<mediated-schema>
  <sources>
    <source id="biblio" url="biblio.xml" />
    <source id="biblio_lirmm" url="biblio_lirmm.xml" />
    <source id="librairie" url="librairie.xml" />
  </sources>
  <extracted-data>
    <source-pattern name="sp_auteurs_biblio" ... />
    <source-pattern name="sp_auteurs_biblio_lirmm" ... />
    <source-pattern name="sp_livres" ... />
    <source-pattern name="f_auteurs" ... />
    <source-pattern name="j_livres_lirmm" ... />
  </extracted-data>
  <views>
    <view name="v_auteurs" ... />
    <view name="v_livres_lirmm" ... />
  </views>
</mediated-schema>
```

FIG. 4.3 – Spécification d'un système d'intégration.

L'élément `sources` contient les trois sources de données qui sont intégrées par le système. Pour chacune de ces sources, l'élément `source` définit un identifiant et contient l'url associée.

Les éléments `extracted-data` et `views` contiennent respectivement la spécification des données à extraire et des vues qui constituent le système d'intégration. Pour des contraintes d'espace, nous

n'avons pas représenté dans la figure le contenu de la spécification des données à extraire et des vues. Tous ces éléments ont été présentés dans les figures précédentes.

4.2.2 Construction du schéma médiateur

La structure spécifiée par le résultat d'une vue permet de générer un schéma du résultat. Les règles de construction de ce schéma ont été présentées plus haut (§ 3.6.3). Le schéma médiateur étant défini comme un ensemble de vues, nous allons utiliser les schémas de ces vues pour le construire.

Le schéma médiateur spécifié par des vues VIMIX est défini comme la collection des DTDs validant les résultats de ces vues. Certains éléments peuvent avoir été définis de manière différente dans les différentes vues du système d'intégration. Par exemple, les deux vues que nous avons spécifiées dans le schéma médiateur présenté plus haut, définissent toutes les deux un élément `auteur` dont la structure est différente. Pour spécifier les éléments et les attributs définis par les différentes DTDs de manière non ambiguë, nous utilisons le mécanisme proposé par le W3C pour définir des espaces de nominaux [Nam99] (*namespaces*).

La DTD validant les données intégrées par notre système est construite de la manière suivante. La racine de cette DTD est arbitrairement un élément `integrated-data`. Elle est composée des sous-éléments représentant les vues spécifiées par le système d'intégration. Chacun de ces éléments possèdent un attribut `xmlns` qui permet de définir un espace de noms. Cet espace nominal porte le nom de la vue et utilise la DTD correspondante. Le nom d'une vue étant un identifiant, cette stratégie garantit qu'il n'y aura pas de définition ambiguë.

Exemple de données intégrées

Nous allons illustrer la structure des données intégrées à l'aide de notre langage de définition de vues. Pour cela, considérons le schéma médiateur qui a été spécifié précédemment (Figure 4.3). La Figure 4.4 présente la structure des données intégrées par cette spécification.

L'élément `integrated-data` défini dans la DTD est composé de deux sous-éléments représentant les deux vues spécifiées par le schéma médiateur : `v_auteurs` et `v_livres_lirmm`. Ces deux sous-éléments possèdent chacun un attribut `xmlns` qui définit un domaine nominal pour la vue. Ce domaine nominal est spécifié par la DTD construite à partir du schéma de la vue correspondante : `v_auteurs.dtd` et `v_livres_lirmm.dtd`. De cette manière, les données intégrées peuvent utiliser de manière non ambiguë l'élément `auteur` en le préfixant par le nom de la vue.

4.2.3 Une approche mixte *GAV / LAV*

L'approche consistant à définir le schéma médiateur comme un ensemble de vues sur les sources est appelée *GAV*. Elle s'oppose à l'approche *LAV* qui définit les sources comme des vues sur le schéma médiateur. Ces deux approches sont présentées plus en détail dans le chapitre 2 ((§ 2.3.2, page 19).

```
<?xml version="1.0" encoding="ISO8859_1" ?>
<!-- dtd definition -->
<!DOCTYPE integrated-data [
<!ELEMENT integrated-data (v_auteurs, v_livres_lirmm) >
<!ELEMENT v_auteurs (#PCDATA) >
<!ELEMENT v_livres_lirmm (#PCDATA) >
]>
<!-- integrated data -->
<integrated-data>

<!-- integrated data of view v_auteurs -->
<v_auteurs xmlns:v_auteurs="v_auteurs.dtd">
<v_auteur:auteur nom="Baril" prenom="Xavier" />
<v_auteur:auteur nom="Bellahsène" prenom="Zohra" />
...
</v_auteurs>

<!-- integrated data of view v_livres_lirmm -->
<v_livres_lirmm xmlns:v_livres_lirmm="v_livres_lirmm.dtd">
<v_livres_lirmm:auteur nom="Baril" nb-livres="1" prix-moyen="49.99">
<v_livres_lirmm:livre>Designing and Managing an XML Warehouse</v_livres_lirmm:livre>
</v_livres_lirmm:auteur>
...
</v_livres_lirmm>

<!-- end of integrated data -->
</integrated-data>
```

FIG. 4.4 – Structure des données intégrées.

Nous avons retenu une approche de type *GAV* car elle simplifie le traitement des requêtes posées sur le schéma médiateur. En revanche, la prise en compte de l'évolution des sources (ajout de nouvelles sources ou évolution du schéma des sources) est généralement plus difficile à traiter qu'avec une approche *LAV*.

Cependant, la spécification d'un schéma médiateur avec VIMIX permet de traiter assez facilement l'ajout de nouvelles sources. Pour cela, notre langage de définition de vues permet de simuler l'approche *LAV*. En effet, les motifs sur les sources (`source-pattern`) sont des vues sur les sources locales. Le schéma médiateur est défini par des vues qui utilisent un graphe de *mappings* composé de nœuds de type `source-pattern`, `fragment` et `join`. Les feuilles de ce graphe sont les motifs sur les sources. Ces motifs permettent donc d'intégrer des sources de données dans le graphe de *mappings* en les définissant comme des vues.

L'ajout d'une nouvelle source est réalisé en ajoutant un motif sur cette source. Si ce motif est utilisé par un fragment existant, le schéma médiateur du système d'intégration ne sera pas modifié, comme avec une approche *LAV*.

Pour ces raisons, nous pouvons qualifier notre approche de mixte, car elle offre les avantages des deux types d'approche *GAV* et *LAV* :

- en spécifiant le schéma médiateur comme une collection de vues,
- en permetant d'ajouter des sources de données sans modifier le schéma médiateur.

De plus, lorsque les vues spécifiées avec VIMIX sont matérialisées, nous proposons des algorithmes de propagation des mises à jour. Ces algorithmes permettent de maintenir incrémentalement les vues matérialisées lorsque les sources évoluent. Les problèmes liés à la matérialisation des vues sont présentés dans le chapitre suivant.

4.3 Mécanisme d'aide pour la spécification de motifs sur les sources

Dans cette section, nous présentons un mécanisme d'aide qui facilite la spécification des motifs. La principale difficulté pour spécifier un motif sur une source de données est de connaître sa structure. Tout d'abord, cette structure peut être irrégulière. Ensuite, le concepteur du système d'intégration ne peut avoir qu'une connaissance partielle de la structure d'une source de données à intégrer.

Afin de faciliter la spécification de motifs sur les sources, nous proposons un **mécanisme d'aide**. Ce mécanisme permet au concepteur du système d'intégration de découvrir la structure d'une source pendant la spécification d'un motif sur cette source. De plus, ce mécanisme permet au concepteur de découvrir seulement partiellement la structure d'une source, c'est à dire sur la partie de la source qu'il est en train de traiter.

Cette section est organisée de la manière suivante. Tout d'abord, nous présentons le principe de fonctionnement du mécanisme d'aide. Afin de découvrir la structure d'une source de données, on peut utiliser deux sources d'information : la DTD si elle existe ou une extension du concept de *dataguide*. Enfin, nous discutons les différences entre ces deux solutions.

4.3.1 Fonctionnement du mécanisme d'aide

Pour spécifier un motif sur une source, on définit des **chemins** dans cette source. Ces chemins permettent de localiser des nœuds de la source. Ces nœuds seront ensuite recherchés pour extraire les données du motif. La difficulté, lorsqu'on a une connaissance partielle de la structure d'une source de données est de définir des chemins qui permettent de localiser des données. En effet, la spécification d'un chemin implique la connaissance de la structure de la source de données.

Un motif sur une source est composé d'une imbrication d'axes de recherche (*search-axis*) et de spécification de nœuds (*source-node*). Cette imbrication permet de définir des chemins sur une source de données. Pour représenter ces chemins nous allons utiliser la notation suivante.

- Pour les fonctions de recherche :
  - `/ :children`,
  - `/*` : `children-comp`,
  - `//` : `descendants`,
  - `//*` : `descendants-comp`.
- Pour les nœuds :
  - `[#type#]reg-expression`.
- Un chemin est composé d'une succession de fonctions de recherche et de nœuds qui représente l'imbrication *search-axis* et *source-node* de la spécification du motif.

Cette notation est inspirée de XPath [XPa99], le langage de localisation de données XML. Cependant, notre modèle de données permet de prendre en compte les liens de référence, et propose des fonctions de recherche différentes de celles de XPath. Nous avons différencié la notation pour les axes `children / children-comp` et `descendants / descendants-comp` à l'aide d'une étoile : "`*`". De plus, la spécification du type d'un nœud n'est pas obligatoire, alors qu'elle est implicite avec XPath (`element` et `@attribut`). Enfin, le nœud contextuel du premier axe de recherche d'un chemin est le nœud racine de la source de données.

Exemple Considérons le motif `sp_auteurs_biblio` présenté dans la Figure 3.10. Les chemins définis par ce motif sont les suivants :

- `/#element#auteur`,
- `/#element#auteur/#element#nom`,
- `/#element#auteur/#element#prenom`.

Pour aider le concepteur du système d'intégration de données, le mécanisme d'aide fonctionne de la manière suivante. Pour chaque axe de recherche, le mécanisme d'aide doit fournir une liste des nœuds qui peuvent être candidat pour l'évaluation. En fait, le mécanisme d'aide doit permettre de compléter les chemins à partir d'une fonction de recherche.

Exemple (suite) Considérons le chemin `/#element#auteur/`. Ce chemin est incomplet, car il se termine par une fonction de recherche (`/` représente la fonction `children`). Le mécanisme d'aide doit proposer une liste de nœuds qui permettent de compléter le chemin. Les nœuds possibles de compléter le chemin appartiennent à la liste de suivante :
`<#attribute#auteur, #element#nom, #element#prenom, #element#email, #element#web>`

4.3.2 Aide basée sur une DTD

Une DTD est une grammaire qui décrit la structure d'un document XML. Lorsqu'une source de données possède une DTD, on peut utiliser cette DTD pour aider le concepteur à découvrir la structure de la source.

Pour cela, on utilise une représentation simplifiée de la DTD. Cette représentation consiste en un graphe qui décrit l'imbrication des éléments et des attributs. Il n'est pas nécessaire de représenter toutes les informations contenues dans la DTD :

- le fait qu'un sous-élément soit optionnel (`?`), ou qu'il puisse apparaître plusieurs fois (`+` ou `*`) n'est pas pris en compte,
- le parenthésage qui ordonne les sous-éléments et l'opérateur `|` ne sont pas pris en compte,
- la déclaration indiquant la présence d'un attribut (`#REQUIRED` ou `#IMPLIED`) n'est pas prise en compte.

Ce graphe possède une racine et deux types de nœuds pour représenter les éléments et les attributs. Bien que notre modèle de données possède deux types de liens (liens de composition et liens de référence), la DTD ne permet de représenter que les liens de composition. Lorsqu'un nœud attribut est de type `IDREF(S)` il peut référencer n'importe quel nœud élément possédant un identifiant (nœud attribut de type `ID`).

Exemple Soit la DTD qui valide les données de la source `biblio` présentée dans la Figure 3.2. Le graphe contenant la représentation simplifiée de cette DTD est présenté dans la Figure 4.5. Les nœuds

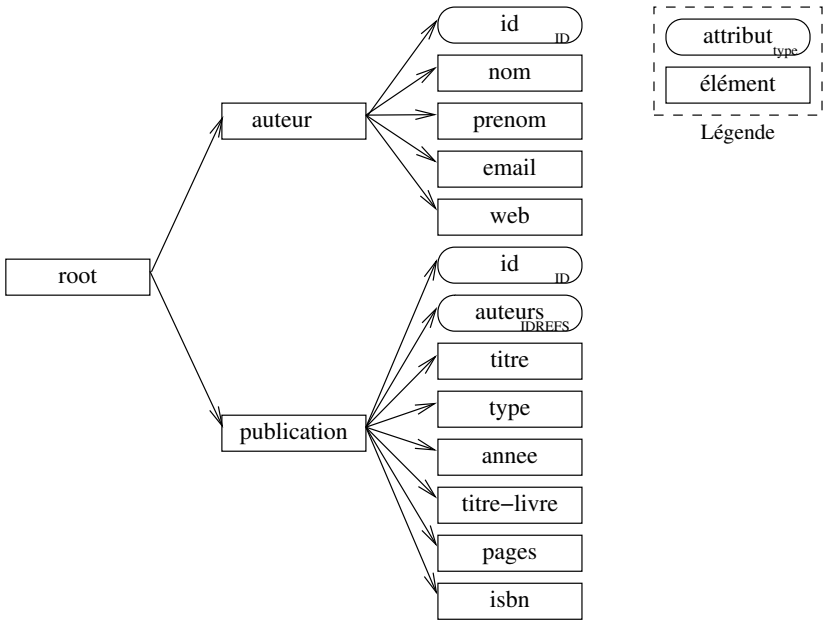

FIG. 4.5 – Représentation simplifiée de la DTD de la source `biblio`.

représentant des éléments sont dans des boites rectangulaires, tandis que les nœuds représentant des

attributs sont dans des boites avec les bords arrondis. Le nœud racine possède deux nœuds fils qui représentent les éléments `auteur` et `publication`. Dans le graphe tous les attributs (avec leur type) et les sous-éléments possible de ces nœuds sont représentés.

A partir d'un chemin incomplet (se terminant par une fonction de recherche), le mécanisme d'aide basé sur la DTD fournit une liste de nœuds possibles pour compléter le chemin. La construction de cette liste s'effectue en deux étapes.

1. Tout d'abord, on cherche dans le graphe représentant la DTD les nœud pouvant être le dernier nœud du chemin incomplet. Un nœud du chemin incomplet peut correspondre à plusieurs nœuds dans le graphe représentant la DTD, du fait de l'utilisation des fonctions de recherche `descendants` et `descendants-comp`. Par exemple, pour le chemin `//#attribute#id`, les nœuds correspondant dans le graphe représentant la DTD sont les nœuds représentant l'attribut `id` de l'élément `auteur` et de l'élément `publication`.
2. Pour chacun des nœuds trouvés lors de la première étape, on applique la fonction de recherche du chemin incomplet pour construire la liste des nœuds pouvant compléter le chemin. Lorsqu'un nœud du graphe représentant la DTD est de type `IDREF(S)`, ces fils sont tous les nœuds élément qui possèdent un identifiant. Par exemple, pour le chemin :
   `/#element#publication/#attribute#auteurs/`
   le mécanisme d'aide proposera la liste suivante :
   `<#element#auteur, #element#publication>`

4.3.3 Aide basée sur un *dataguide*

Un *dataguide* est un résumé de la structure d'une source de données semistructurées. Le concept de *dataguide* a été proposé à l'origine pour faciliter la formulation de requête dans le système Lore [GW97]. Ce système est un SGBD manipulant des données semistructurées au format OEM [MAG+97, GMW00, GMW99, AQM+97]. Nous avons adapté la définition du *dataguide* à notre modèle de données pour XML, afin de proposer un mécanisme d'aide pour la spécification de motifs sur les sources.

Nous avons défini un *dataguide* comme un résumé d'un graphe de données XML où chaque chemin apparaît une seule fois. Les nœuds textes ne sont pas représentés dans le *dataguide*. Le *dataguide* est donc un graphe (ayant une racine), qui possède deux types de nœuds (élément et attribut) et deux types de liens (composition et référence).

Exemple Soit la source de données `biblio` présentée dans la Figure 3.3. Le *dataguide* de cette source est présenté dans la Figure 4.6.

Les nœuds représentant des éléments sont dans des boites rectangulaires, tandis que les nœuds représentant des attributs sont dans des boites avec les bords arrondis. Les liens de composition sont représentés par des traits pleins, tandis que les liens de référence sont représentés par des traits pointillés.

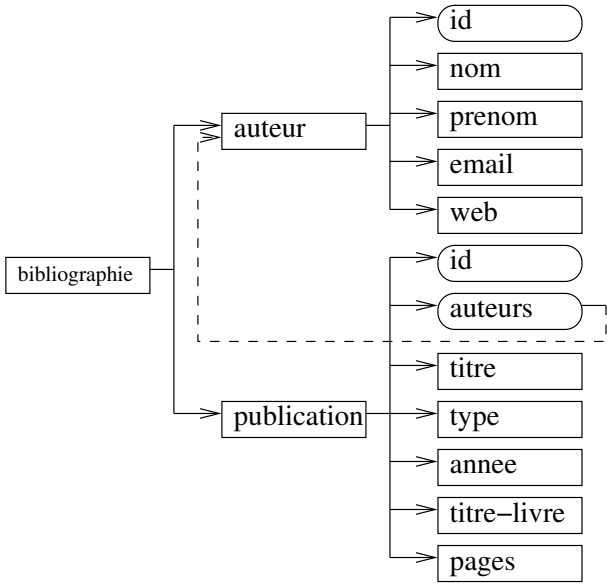

FIG. 4.6 – *Dataguide* de la source de données `biblio`.

Comme pour le mécanisme d'aide basé sur la DTD, la construction de la liste de nœuds pouvant compléter le chemin s'effectue en deux étapes.

1. Tout d'abord, on cherche dans le *dataguide* les nœuds pouvant être le dernier nœud du chemin incomplet.
2. Pour chacun des nœuds trouvés lors de la première étape, on applique la fonction de recherche du chemin incomplet pour construire la liste des nœuds pouvant compléter le chemin. La fonction de recherche peut s'exécuter comme dans un graphe de données XML, en utilisant les liens de composition et/ou de référence. Par exemple, pour le chemin :
   `/#element#publication/#attribute#auteurs/`
   le mécanisme d'aide proposera la liste suivante :
   `<#element#auteur>`

4.3.4 Comparaison des deux mécanismes d'aide

Les deux mécanismes d'aide que nous proposons sont complémentaires. Ils permettent tous les deux de faciliter le travail du concepteur d'un système intégrant des sources XML. Il existe certaines différences importantes entre ces deux mécanismes que nous allons présenter ici.

Disponibilité Lorsque les sources de données ne possèdent pas de DTD, on ne peut utiliser que le mécanisme d'aide basé sur un *dataguide*.

Temps de calcul La construction du graphe qui représente la DTD d'une source de données est peu coûteuse en temps. En effet, la taille d'une DTD est souvent beaucoup plus petite que celle de la source

de données, car elle ne contient que la grammaire qui permet de décrire sa structure. La construction du *dataguide* est plus coûteuse car elle nécessite de *parser* tout le document contenant la source de données. Cette opération peut être longue si la taille du document est importante.

Expressivité Les DTDs ne permettent pas de spécifier le type des éléments cibles d'un attribut de type `IDREF(S)`. Le graphe qui représente une DTD contient des liens de composition mais ne permet pas de représenter les liens de référence. Le *dataguide*, puisqu'il est basé sur notre modèle de données pour XML permet de représenter les liens de composition et de référence.

Contenu Lorsque le mécanisme d'aide est basé sur la DTD, il peut proposer des nœuds qui n'existent pas dans la source de données. En effet, il est possible que certaines parties de la structure décrite par la DTD ne soient pas instanciées dans la source de données. A l'inverse, le *dataguide* propose seulement des nœuds qui existent dans la source de données.

4.4 Conclusion

Les vues définies avec VIMIX permettent d'**intégrer** des données provenant de sources XML multiples et hétérogènes. Le **schéma médiateur** d'un système d'intégration basé sur VIMIX est défini comme une collection de vues. Il est construit en utilisant les schémas générés pour les vues qui le définissent.

L'intégration des données s'effectue en trois étapes. La première consiste à définir des motifs sur les sources pour spécifier les données à extraire. Ensuite, on construit un graphe de *mappings* qui permet de restructurer les données à extraire en utilisant des opérations d'union et de jointure. Enfin, les données spécifiées dans ce graphe de *mappings* sont utilisées pour spécifier des vues. Comparé aux approches GAV et LAV, notre approche peut être qualifiée de **mixte** car elle profite de certains des avantages de ces deux approches. En effet, la notion de fragment permet d'ajouter facilement des sources de données.

La principale difficulté pour définir des motifs sur les sources est de connaître leur structure. Afin de faciliter cette tâche, nous proposons un **mécanisme d'aide** permettant au concepteur d'un motif de découvrir la structure d'une source, pour les nœuds qu'il est en train de spécifier. Pour cela, nous utilisons deux sources d'information. La DTD de la source si elle existe permet de découvrir sa structure. Nous avons également adapté le concept de *dataguide* à notre modèle de données pour XML afin de découvrir la structure d'une source.

Le chapitre suivant présente notre proposition pour matérialiser les vues définies avec VIMIX. Le stockage des données est basé sur une méthode de méta modélisation utilisant un SGBD relationnel.

Cette méthode permettra d'utiliser des algorithmes de maintenance, qui permettent de gérer la mise à jour des données et l'évolution des sources de manière incrémentale.

Chapitre 5

Matérialisation de vues VIMIX

5.1 Introduction

Dans ce chapitre, nous présentons notre solution pour matérialiser les vues VIMIX dans un SGBD relationnel. A un certain niveau d'abstraction, les entrepôts de données sont généralement définis comme une collection de vues matérialisées [Wid95]. La matérialisation des vues permet donc de construire un entrepôt de données XML à partir d'un système d'intégration défini avec VIMIX. Le schéma médiateur de cet entrepôt est défini à partir des schémas des vues, comme cela est présenté dans le chapitre 4 (§ 4.2.2, page 70).

L'entrepôt de données XML ainsi défini est stocké en utilisant un SGBD relationnel. L'architecture de stockage repose sur un schéma générique pour stocker les données XML des sources. Les *mappings* qui spécifient les données extraites par des motifs sur les sources, des fragments et des jointures sont des méta données qui sont stockées dans des tables. Ces tables sont organisées en un graphe de *mappings*. De plus, nous montrons que notre méthode de stockage facilite la maintenance incrémentale de l'entrepôt.

Comme l'entrepôt est stocké dans un SGBD relationnel, on accède aux données avec des requêtes SQL. Pour cette raison, les requêtes posées sur les données XML décrites par le schéma médiateur doivent être réécrites en requêtes SQL. L'intérêt de cette approche est double. Elle permet de profiter des performances des processeurs de requêtes SQL. La méthode de méta modélisation utilisée pour le stockage facilite le traitement des requêtes interrogeant la structure des données.

La Figure 5.1 illustre l'**interface** qui permet d'interroger des données XML stockées dans un SGBD relationnel. D'un point de vue fonctionnel, cette interface définie par le schéma médiateur donne l'illusion d'un système "tout XML". En effet, elle accepte en entrée des requêtes XML et fournit en sortie un résultat XML. Cette interface est constituée de deux composants, chargés de la **réécriture** des requêtes et du **formatage** du résultat. Nous allons illustrer leur rôle en décrivant le traitement d'une requête.

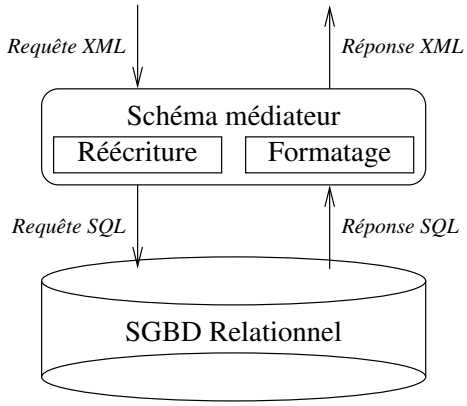

FIG. 5.1 – Interface pour l'interrogation du schéma médiateur.

1. Lorsqu'une requête XML est posée au système, le composant de réécriture se charge de la transformer en SQL. La requête SQL est ensuite envoyée au SGBD relationnel qui va l'exécuter avec son propre gestionnaire de requêtes.
2. Lorsque l'exécution de la requête SQL est terminée, le SGBD relationnel renvoie son résultat au composant chargé du formatage. Ce composant permet de construire les données XML qui répondent à la requête initialement posée.

L'interrogation de données XML stockées dans un SGBD relationnel a déjà été étudiée [MFK01, YASU01, IDD01]. Les requêtes XML sont réécrites en SQL en appliquant des **règles de transformation**. Ces règles permettent de construire une requête SQL qui sera exécutée sur le schéma relationnel utilisé pour stocker les données XML. Nous avons défini des règles de transformation qui permettent de réécrire des requêtes sur le schéma générique que nous utilisons pour stocker les données XML de l'entrepôt. Le langage d'interrogation et les règles de réécriture de requêtes sont présentées en annexe (E, page 165).

Notre architecture de stockage utilise des tables relationnelles pour stocker les *mappings* des données qui correspondent aux vues. La définition des vues permet de créer de nouveaux éléments, à partir des données XML des sources. Ces nouveaux éléments ne sont pas stockés dans l'entrepôt, seules les données qui permettent de les construire sont matérialisées. Les requêtes sont posées sur le schéma médiateur de l'entrepôt. Le traitement des requêtes doit utiliser les *mappings* des vues pour reconstruire les éléments XML définis par les vues : cela permet d'exploiter la structure des vues pour interroger l'entrepôt. En effet, il n'est pas nécessaire de parcourir toutes les données d'une vue pour répondre à une requête. Le schéma médiateur permet de déterminer quels sont les *mappings* nécessaires pour construire les éléments des vues qui répondent à la requête.

Ce chapitre est organisé comme suit.

– Dans la section 5.2, nous présentons l'architecture et la méthode utilisées pour le stockage de vues VIMIX, en utilisant un SGBD relationnel. Les données XML des sources sont stockées en utilisant un schéma générique. Les méta données spécifiant les données extraites sont stockées dans des tables organisées en un graphe de *mappings*.
– Dans la section 5.3, nous présentons les algorithmes qui permettent de maintenir les vues. Nous traitons les problèmes de rafraîchissement des données et d'évolution des vues. Les algorithmes proposés permettent de maintenir incrémentalement les vues, en propageant les opérations de mise à jour dans le graphe de *mappings*.
– Dans la section 5.4, nous présentons la méthode qui permet de construire le résultat des vues. Pour cela, nous utilisons des algorithmes qui utilisent la spécification des vues et les données stockées dans le SGBD relationnel.

5.2 Stockage de vues VIMIX

5.2.1 Approches existantes pour le stockage de données XML

On distingue principalement deux stratégies possibles pour le stockage de données XML : le **stockage à plat** et la **méta modélisation**. Il existe également une troisième famille de solutions qui sont appelées **approches hybrides**. Ces approches consistent à combiner l'utilisation des deux stratégies précédentes.

Stockage à plat

Avec cette méthode de stockage, les données XML sont stockées en utilisant leur forme textuelle. C'est la méthode la plus simple à mettre en oeuvre, car il suffit d'utiliser un système de fichiers, ou le type `BLOB` d'un SGBD pour stocker les documents dans une base de données.

Cette méthode est très efficace quand on essaye de retrouver la totalité ou des grosses parties contiguës d'un document XML. L'inconvénient principal de cette méthode est la nécessité de *parser* le document pour découvrir sa structure : cela a pour conséquence de ralentir le traitement des requêtes.

Méta modélisation

Cette méthode de stockage consiste à modéliser les données XML en utilisant le modèle de données d'un SGBD conventionnel. Les données XML sont alors stockées dans le SGBD cible en utilisant des **règles de transformation**. Ces règles permettent de transformer les données XML pour les représenter en utilisant le modèle de données du SGBD cible. Pour cela, on peut utiliser un SGBD relationnel [BB03a, FK99, MFK+00, YASU01, SGN99] ou objet [AJEM02].

Cette méthode est très efficace lorsqu'on veut formuler des requêtes qui utilisent la structure des données stockées. En effet, les données ont déjà été analysées lors de leur transformation pour être

stockées dans le SGBD cible. La structure ainsi stockée permet de répondre efficacement aux requêtes. L'inconvénient principal de cette méthode réside dans les transformations qui sont nécessaires pour stocker et reconstruire les données des documents XML. Lorsque les documents XML à stocker sont volumineux, cette phase de transformation est coûteuse.

Approches hybrides

On peut aussi combiner les deux approches précédentes, il existe deux manières de le faire. La première est redondante, elle consiste à stocker les données en utilisant les deux méthodes. Cela permet une interrogation rapide des documents ainsi stockés, mais naturellement les mises à jour sont ralenties et l'espace de stockage est loin d'être optimal car toutes les données sont dupliquées.

La deuxième méthode consiste à utiliser une approche mixte : à partir d'un certain niveau de granularité appelé seuil, les données sont stockées à plat alors qu'au dessus de ce niveau elles sont stockées dans un SGBD en utilisant la méta modélisation. Cette approche est utilisée par le système NATIX [KM99, KM00] qui permet de stocker de gros volumes de données dans Xylème [MAA+00].

Discussion

Le Tableau 5.1 compare les points forts et les faiblesses du stockage à plat et de la méta modélisation dans le contexte du stockage d'une collection de documents XML. Pour le stockage à plat, on considère qu'on utilise un système de gestion de fichier. Nous n'avons pas introduit les méthodes hybrides car elles proposent un compromis des points forts et des points faibles de chacune des méthodes présentées. Les différents axes de comparaison sont :

- l'**ajout d'un document** XML dans l'entrepôt,
- la **suppression d'un document** XML dans l'entrepôt,
- l'exécution d'une **requête** portant sur les données des différents documents de l'entrepôt,
- la **consultation** d'un document de l'entrepôt.

La lecture de ce tableau nous amène à la constatation suivante : il n'y a pas de méthode de stockage idéale, qui surpasserait les autres dans tous les cas d'utilisation. Il faut donc choisir la méthode de stockage à utiliser en fonction des cas d'utilisation du problème.

Concernant la méta modélisation, on peut distinguer deux sortes de schémas.

1. Les schémas **génériques** qui peuvent être utilisés pour toute instance de données XML.
2. Les schémas **dépendant des données** qui doivent être générés pour chaque instance de données à stocker.

Intuitivement, on constate que l'utilisation d'un schéma générique peut s'avérer plus simple lorsque les données à stocker proviennent de documents hétérogènes. En effet, si on utilise un schéma dépendant des données, il faudra générer un schéma de stockage pour chaque document.

	Stockage à plat	Méta modélisation
Ajout document	**Point fort** : il suffit de copier un fichier	**Point faible** : il faut analyser le document avant de pouvoir le stocker
Suppression document	**Point fort** : il suffit de supprimer un fichier	**Moyen** : il faut exécuter une requête qui supprime tous les nœuds de ce document dans la base
Requête	**Point faible** : il faut analyser les fichiers contenant les documents pour répondre à la requête	**Point fort** : il suffit de réécrire la requête et de la poser au SGBD cible, ce qui garantit un bon niveau de performance
Consultation	**Point fort** : il suffit de lire un fichier	**Point faible** : il faut reconstruire le document

TAB. 5.1 – Comparaison des stratégies de stockage de données XML.

Parmi les stratégies de méta modélisation utilisant un SGBD relationnel que nous avons présentées, il n'y en a aucune qui se positionne comme étant meilleure que les autres dans tous les cas d'utilisation. Il s'avère donc judicieux de choisir le schéma de méta modélisation en fonction :

- des requêtes qui seront posées au système [BFRS02, BFH+02],
- de la morphologie des données à stocker.

Les données XML peuvent avoir une forme très variée car elles permettent de représenter de nombreux formats et types de données. En effet, une source de données XML peut avoir été obtenue à partir de données stockées dans un SGBD relationnel. Dans ce cas, il y a de fortes chances pour que la structure des données soit assez régulière. A l'opposé, une source de données XML peut avoir été construite par des utilisateurs différents qui n'ont pas utilisé le même vocabulaire : les données seront alors moins structurées que dans le cas précédent. La morphologie d'un document XML peut être caractérisée par de nombreux paramètres. On peut considérer par exemple la profondeur de l'arbre des éléments, le ratio entre le nombre de types d'éléments différents et le nombre total d'éléments, le nombre moyen d'attributs par élément, la taille moyenne du nom d'un élément, etc. A notre connaissance il n'y a pas de travaux prenant en compte ces paramètres.

5.2.2 Architecture pour le stockage de vues VIMIX

L'architecture que nous avons définie pour le stockage des données de notre entrepôt utilise un SGBD relationnel. La motivation principale pour ce choix est qu'il permet d'accélérer le traitement des requêtes sur les données de l'entrepôt : la méta modélisation permet de profiter de la structure des données pour les interroger et la réécriture des requêtes en SQL permet de profiter des performances élevées des SGBD relationnels actuels.

La Figure 5.2 présente l'architecture générale que nous avons proposé pour le stockage d'un entrepôt de données défini comme un ensemble de vues VIMIX. Cette figure illustre la séparation entre le stockage

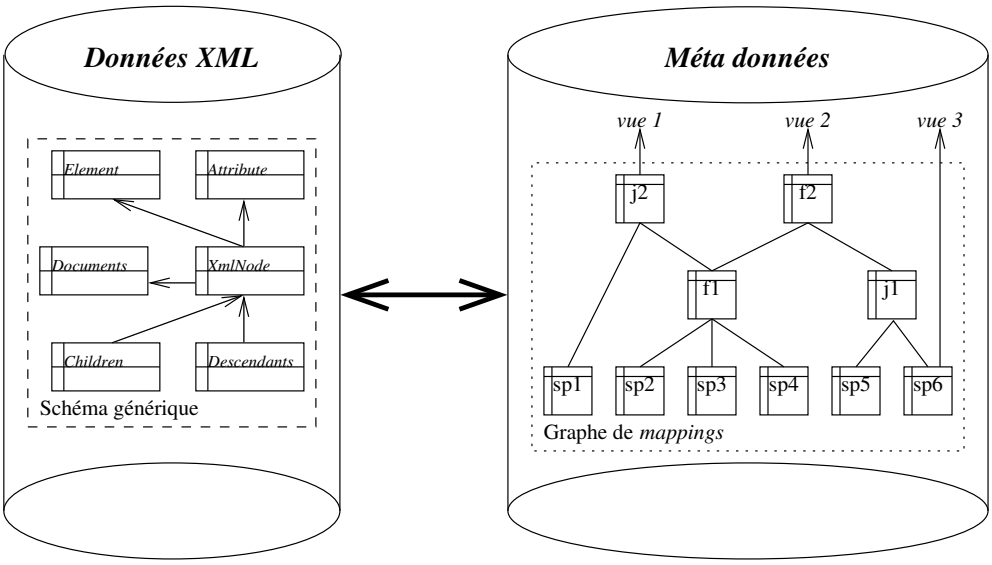

FIG. 5.2 – Architecture de stockage de notre entrepôt de données.

des données XML et des méta données. La partie gauche de la figure montre que les données XML sont stockées en utilisant un schéma générique (présenté dans § 5.2.3. La partie droite de la figure montre l'organisation des méta données. Nous appelons méta données les données qui contiennent les correspondances (*mappings*) entre les données des sources et les données des vues. Ces *mappings* sont organisés en un graphe appelé graphe de *mappings* (présenté dans § 5.2.4. Les nœuds de ce graphe sont des tables permettant de stocker les données spécifiées par des motifs sur les sources, des fragments et des jointures.

Notre architecture de stockage offre les avantages suivants.

- Nous séparons le stockage des données des sources de celui des méta données. Cela permet de ne pas introduire de données XML redondantes dans l'entrepôt et a pour effet :
  - d'améliorer l'espace de stockage,
  - de faciliter la maintenance des données.
- Nous utilisons un mécanisme d'identifiant (pour stocker les méta données) qui permet de gérer incrémentalement la maintenance des données.

Comme les autres approches utilisant la méta modélisation pour stocker des données XML, une phase de traitement est nécessaire pour stocker les données dans le SGBD relationnel. Cependant, ce surcout est compensé par la matérialisation qui permet d'accélerer le traitement des requêtes. Nous considérons que le rôle principal de l'entrepôt est de répondre à des requêtes.

5.2.3 Schéma générique pour le stockage de données XML

Nous présentons ici la méta modélisation que nous proposons pour stocker les données des sources XML dans l'entrepôt. Nous utilisons un schéma générique qui permet de stocker dans un SGBD relationnel des nœuds provenant de sources XML. La Figure 5.3 décrit ce schéma en utilisant les règles typographiques suivantes :

- les noms des colonnes constituant la clé primaire sont soulignés,
- les noms des colonnes représentant des clés étrangères sont en italique.

Table	**Colonnes**
`Document`	`docID`, `url`
`Element`	`elemID`, `name`
`Attribut`	`attID`, `name`
`XmlNode`	`nodeID`, *`elemID`*, *`attID`*, `value`, *`docID`*
`Children`	*`fatherID`*, *`childID`*, `rank`
`Descendants`	*`fatherID`*, *`childID`*, `rank`

FIG. 5.3 – Schéma générique pour le stockage de données XML.

La table `Document` contient les `urls` des sources de l'entrepôt. Le schéma générique permet de stocker des nœuds provenant des sources, sans stocker toutes les données de cette source. Pour chaque source (ou document) on a simplement besoin de connaître son identifiant et son url, sans se préoccuper de la racine du document qui n'est pas nécessairement stockée dans l'entrepôt.

Les tables `Element` et `Attribut` sont des dictionnaires des éléments et des attributs stockés dans l'entrepôt. Elles contiennent un code qui permet d'identifier l'élément ou l'attribut ainsi que son nom. Nous verrons dans le chapitre suivant que ces dictionnaires permettent d'accélérer les requêtes lorsqu'elles portent sur un nom d'élément ou d'attribut.

La table `XmlNode` permet de stocker les nœuds des sources. Chaque nœud possède un identifiant : `nodeID`. Notre modèle de données considère trois types de nœuds : *Element*, *Attribute* et *Text* qui représentent respectivement des éléments, des attributs et du texte dans un document XML.

Les colonnes `elemID` et `attID` permettent de connaître le type d'un nœud de la table :

- Si `elemID` est non nul[1], le nœud est de type élément et `elemID` désigne son nom (dans le dictionnaire des éléments).
- Si `attID` est non nul, le nœud est de type attribut et `attID` désigne son nom (dans le dictionnaire des attributs). La valeur de l'attribut est stockée dans la colonne `value`.

[1] Au sens relationnel du terme (NULL correspond à l'absence de valeur).

– Enfin, si `elemID` et `attID` sont tous les deux nuls, le nœud est de type texte et `value` contient la chaîne de caractères.

Enfin, pour les trois types de nœud la colonne `docID` contient l'identifiant de la source. Ce fonctionnement est résumé par la Table 5.2 : le rôle des colonnes de la table est précisé en fonction du type de nœud représenté.

Type de nœud	**Colonnes**				
	`nodeID`	`elemID`	`attID`	`value`	`docID`
Element	identifiant	nom	NULL	NULL	source
Attribut	identifiant	NULL	nom	valeur	source
Text	identifiant	NULL	NULL	valeur	source

TAB. 5.2 – Rôle des colonnes de la table `XmlNode` en fonction du type de nœud représenté.

La table `Children` contient les liens de composition et de référence entre les nœuds de l'entrepôt. Elle est composée des colonnes suivantes :

– `fatherID` contient l'identifiant du nœud père,
– `childID` contient l'identifiant du nœud fils,
– `rank` contient le rang du fils.

Les identifiants `fatherID` et `childID` sont des clés étrangères de la colonne `nodeID` de la table `XmlNode`.

La table `Descendants` contient les liens de descendance entre les éléments qui sont stockés dans l'entrepôt. Elle est composée des colonnes suivantes :

– `fatherID` contient l'identifiant du nœud père,
– `childID` contient l'identifiant du nœud fils,
– `rank` contient le rang du fils en considérant un parcours en profondeur d'abord.

Les identifiants `fatherID` et `childID` sont des clés étrangères de la colonne `nodeID` de la table `XmlNode`. Les informations contenues dans cette table sont redondantes mais utiles car le langage SQL ne permet pas de les déduire. En effet, il n'est pas possible d'écrire une requête qui fournit tous les descendants d'un nœud à un niveau quelconque à partir des liens père-fils des nœuds d'un arbre[2].

Exemple Considérons la source de données `biblio` (B.1, page 155) présentée dans le chapitre 3 (Figure 3.3, page 41). Cette source contient des données bibliographiques et on souhaite stocker tous les éléments `auteur` dans l'entrepôt. La table `Document` contient alors une seule ligne contenant l'identifiant et l'url de la source :

[2]Le SGBD Oracle propose une clause `connect by` qui permet de parcourir une hiérarchie mais qui ne fait pas partie de la norme SQL.

Document	docID	url
	1	biblio.xml

Les tables `Element` et `Attribute` contiennent les dictionnaires des éléments et des attributs existant dans les nœuds de la source qui sont stockés dans l'entrepôt.

Element	elemID	name
	1	auteur
	2	nom
	3	prenom
	4	email
	5	web

Attribute	attID	name
	1	id

La table `XmlNode` contient tous les nœuds de la source qui sont stockés dans l'entrepôt : les deux nœuds `auteur` de la source `biblio` ainsi que tous leurs descendants.

XmlNode	nodeID	elemID	attID	value	docID
	1	1	NULL	NULL	1
	2	NULL	1	xb	1
	3	2	NULL	NULL	1
	4	NULL	NULL	Baril	1
	5	3	NULL	NULL	1
	6	NULL	NULL	Xavier	1
	7	4	NULL	NULL	1
	8	NULL	NULL	xavier.baril@free.fr	1
	9	4	NULL	NULL	1
	10	NULL	NULL	baril@lirmm.fr	1
	11	5	NULL	NULL	1
	12	NULL	NULL	http ://xavier.baril.free.fr	1
	13	1	NULL	NULL	1
	14	NULL	1	zb	1
	15	2	NULL	NULL	1
	16	NULL	NULL	Bellahsène	1
	17	3	NULL	NULL	1
	18	NULL	NULL	Zohra	1
	19	4	NULL	NULL	1
	20	NULL	NULL	bella@lirmm.fr	1

Les tables `Children` et `Descendants` contiennent les liens les nœuds de la source. Pour des raisons d'espace, nous ne présenterons pas les données de la table `Descendant`.

Children	fatherID	childID	rank
	1	2	1
	1	3	2
	3	4	1
	1	5	3
	5	6	1
	1	7	4
	7	8	1
	1	9	5
	9	10	1
	1	11	6
	11	12	1
	13	14	1
	13	15	2
	15	16	1
	13	17	3
	17	18	1
	13	19	4
	19	20	1

5.2.4 Stockage des méta données

Les méta données sont des données particulières. On peut les définir par le fait qu'elles permettent d'**accéder** à des données ou de **représenter** des données. Cette définition est assez large et désigne des données de nature diverse.

- Le schéma d'une base de données permet de représenter les données qui seront stockées dans un SGBD. Ce schéma permet d'accéder aux données en utilisant généralement un langage de requête. Les données qui expriment le schéma sont donc des méta données. De la même façon, la spécification de l'entrepôt utilise des méta données qui décrivent le schéma médiateur.
- Les SGBD et les entrepôts actuels utilisent souvent des statistiques sur les données : ce sont aussi des méta données. On peut citer par exemple la fréquence d'accès à des tables ou des requêtes, la fréquence de mise à jour des tables, les taux de sélectivité de certains opérateurs, ...
- Dans les systèmes d'intégration, les données des sources doivent être associées au schéma médiateur. Cette association s'effectue à l'aide de règles de correspondance (*mappings*) entre les données des sources et le schéma médiateur. Les données contenant les *mappings* sont des méta données car elles permettent d'accéder aux données des sources.

La spécification d'une vue permet de restructurer les données des sources en utilisant un motif sur une source, un fragment ou une jointure. Ces trois éléments du langage de spécification de vues constituent une **interface** entre les données des sources et la vue. Ce sont donc des méta données

qui permettent d'exprimer des ***mappings*** entre les données des sources et le schéma de la vue. C'est l'organisation de ces données que nous présentons ici.

Le chapitre 3, consacré à la spécification des données à extraire, montre que les motifs sur les sources, les fragments et les jointures peuvent être représentés d'un point de vue logique par des tables relationnelles. Pour matérialiser les vues VIMIX et construire un entrepôt de données, nous allons stocker les méta données des tables relationnelles. Pour chaque motif, fragment et jointure, nous allons construire une table qui stockera les *mappings* permettant de spécifier les données extraites.

Stockage des *mappings* des motifs sur les sources

Pour chaque motif *sp*, nous allons construire une table contenant les *mappings* permettant de spécifier les données extraites. Le schéma de cette table sera de la forme :

$$sp(id, variables_{sp})$$

sp est le nom du motif, défini par l'attribut `name` de sa spécification.

id est un identifiant numérique de type entier, qui utilise l'ordre d'insertion des données dans la table représentant le motif. Cet identifiant permet donc de conserver l'ordre défini par l'extraction des données dans la table contenant les données du motif.

$variables_{sp}$ est l'ensemble des variables spécifiées par le motif *sp*. Chaque colonne correspondant à une variable permet de référencer un nœud de données stocké dans le schéma générique.

Exemple Considérons le motif `sp_auteurs_biblio` (Figure 3.10, page 49) spécifiant les auteurs de la source `biblio`. La table permettant de stocker les *mappings* de ce motif aura le schéma suivant :
`sp_auteurs_biblio(id, nom, prenom)`
Si on considère les identifiants des nœuds XML définis plus haut (§ 5.2.3) pour la source `biblio`, le contenu de la table `sp_auteurs_biblio` sera le suivant :

sp_auteurs_biblio	id	nom	prenom
	1	3	5
	2	15	17

Pour faciliter la lecture, nous présentons également la table en recomposant les nœuds XML qui sont référencés :

id	nom	prenom
1	<nom>Baril</nom>	<prenom>Xavier</prenom>
2	<nom>Bellahsène</nom>	<prenom>Zohra</prenom>

Considérons le motif `sp_auteurs_biblio_lirmm` (C.2, page 159) spécifiant les auteurs de la source `biblio-lirmm`. La table permettant de stocker les *mappings* de ce motif pourra être la suivante (pour des raisons d'espace, les données correspondantes ne sont pas présentées dans le schéma générique) :

sp_auteurs_biblio_lirmm	id	nom	prenom	email
	1	100	101	102
	2	103	104	105

Pour faciliter la lecture, nous présentons également la table en recomposant les nœuds XML qui sont référencés :

id	nom	prenom	email
1	nom="Bellahsène"	prenom="Zohra"	email="bella@lirmm.fr"
2	nom="Hérin"	prenom="Danièle"	email="dh@lirmm.fr"

Considérons le motif `sp_livres` (C.3, page 160) spécifiant les livres de la source `librairie`. La table permettant de stocker les *mappings* de ce motif pourra être la suivante (pour des raisons d'espace, les données correspondantes ne sont pas présentées dans le schéma générique) :

sp_livres	id	titre	auteur	prix
	1	1000	1002	1004
	2	1000	1006	1004

On remarque, que les éléments `titre` et prix sont partagés par les deux lignes de la table car ils correspondent au même élément dans la source `librairie`. Pour faciliter la lecture, nous présentons également la table en recomposant les nœuds XML qui sont référencés :

id	titre	auteur	prix
1	<titre>Designing ...</titre>	<nom>Baril</nom>	<prix>49.99</prix>
2	<titre>Designing ...</titre>	<nom>Bellahsène</nom>	<prix>49.99</prix>

Stockage des *mappings* d'un fragment

Pour fragment f, nous allons construire une table contenant les *mappings* permettant de spécifier les données extraites. Le schéma de cette table sera de la forme :

$$f(id, sid, variables_f)$$

f est le nom du fragment, défini par l'attribut `name` de sa spécification.

id est identifiant numérique de type réel, dont la sémantique des parties entière et décimale est définie comme suit.

- La partie entière contient le numéro d'ordre de la source dont provient la ligne de données du fragment. Ce numéro d'ordre est obtenu par la position de la source dans la liste des sources du fragment.

– La partie décimale contient l'identifiant de la source dont provient la ligne de données. Si cet identifiant est un nombre réel, il est transformé en nombre entier en concaténant les parties entière et décimale. Par exemple, si l'identifiant d'une ligne de la source à insérer dans le fragment est 2.123, la transformation donnera le résultat 2123.
De plus, la partie décimale de la source à la position i est précédée de $n - k$ zéros, avec n et k définis comme suit. 10^n est la borne supérieure minimale du nombre de sources du fragment et 10^k est la borne supérieure minimale de i. Par exemple, si un fragment est défini sur une liste de 11 sources, la borne supérieure minimale du nombre de sources de la forme 10^n est 10^2, donc on a $n = 2$. La partie décimale des identifiants des lignes provenant des sources à la position i, pour $i \in [1..9]$ sera précédée d'un zéro, car la borne supérieure minimale de i est 10^1, soit $k = 1$, on a donc $n - k = 1$. La partie décimale des identifiants des lignes provenant des sources à la position i, pour $i \in [10..11]$ ne sera précédée d'aucun zéro, car la borne supérieure minimale de i est 10^2, soit $k = 2$, on a donc $n - k = 0$. Cette méthode permet de conserver l'ordre qui est défini entre les lignes des tables contenant les sources du fragment.

Cet identifiant permet donc de conserver l'ordre défini lors de l'extraction des données. La partie entière conserve l'ordre entre les listes des sources du fragment. La partie décimale conserve l'ordre défini entre les lignes des sources du fragment.

La colonne `sID` référence l'identifiant de la ligne insérée, dans la source dont elle provient. Cette colonne est de type réel, car elle doit contenir les identifiants des sources de données du fragment, qui peuvent être de type entier ou réel.

$variables_f$ est l'ensemble des variables spécifiées par le fragment f. Chaque colonne correspondant à une variable permet de référencer un nœud de données stocké dans le schéma générique.

Exemple Considérons le fragment `f_auteurs` (Figure 3.13, page 53) spécifiant l'union des auteurs des motifs `sp_auteurs_biblio_lirmm` et `sp_auteurs_biblio`. La table permettant de stocker les *mappings* de ce fragment sera la suivante (en considérant les tables présentées dans l'exemple précédent) :

f_auteurs	id	sid	nom	prenom	email
	1.1	1	101	102	103
	1.2	2	104	105	106
	2.1	1	3	5	NULL

Pour faciliter la lecture, nous présentons également la table en recomposant les nœuds XML qui sont référencés :

id	sid	nom	prenom	email
1.1	1	nom="Bellahsène"	prenom="Zohra"	email="bella@lirmm.fr"
1.2	2	nom="Hérin"	prenom="Danièle"	email="dh@lirmm.fr"
2.1	1	<nom>Baril</nom>	<prenom>Xavier</prenom>	

Stockage des *mappings* d'une jointure

Pour fragment j, nous allons construire une table contenant les *mappings* permettant de spécifier les données extraites. Le schéma de cette table sera de la forme :

$$j(id, lid, rid, variables_j)$$

j est le nom de la jointure, défini par l'attribut `name` de sa spécification.

id est un identifiant numérique de type réél, dont la sémantique des parties entière et décimale est définie comme suit.

- La partie entière contient l'identifiant des données provenant de la partie gauche de la jointure. Si cet identifiant est un nombre réel, il est transformé en nombre entier.
- La partie décimale contient l'identifiant des données provenant de la partie droite de la jointure. Si cet identifiant est un nombre réel, il est transformé en nombre entier.

lid est un identifiant numérique de type réel, qui référence l'identifiant de la ligne utilisée pour calculer la partie gauche de la jointure. Cette colonne est de type réel, car elle doit contenir les identifiants des deux sources de la jointure qui peuvent être de type entier ou réel.

rid est un identifiant numérique de type réel, qui référence l'identifiant de la ligne utilisée pour calculer la partie droite de la jointure. Cette colonne est de type réel, car elle doit contenir les identifiants des deux sources de la jointure qui peuvent être de type entier ou réel.

$variables_j$ est l'ensemble des variables spécifiées par la jointure j. Chaque colonne correspondant à une variable permet de référencer un nœud de données stocké dans le schéma générique.

Exemple Considérons la jointure `j_livres_lirmm` (Figure 3.13, page 53) spécifiant le croisement des informations concernant des auteurs et des livres. La table permettant de stocker les *mappings*de cette jointure sera la suivante (en considérant les tables présentées dans l'exemple précédent). Pour des raisons d'espace, nous n'avons pas représenté les préfixes des variables de la jointure.

j_livres_lirmm	id	lid	rid	titre	auteur	prix	nom	prenom	email
	1.21	1	2.1	1000	1002	1004	3	5	NULL
	2.11	2	1.1	1000	1006	1004	101	102	103

Pour faciliter la lecture, nous présentons également la table en recomposant les nœuds XML qui sont référencés. Pour des raisons d'espace, nous avons inversé l'orientation de la figure (les lignes de la table sont représentées par les colonnes de la figure.

id	1.21	2.11
lid	1	2
rid	2.1	1.1
titre	<titre>Designing ...</titre>	<titre>Designing ...</titre>
auteur	<nom>Baril</nom>	<nom>Bellahsène</nom>
prix	<prix>49.99</prix>	<prix>49.99</prix>
nom	<nom>Baril</nom>	nom="Bellahsène"
prenom	<prenom>Xavier</prenom>	prenom="Zohra"
email		email="bella@lirmm.fr"

Graphe de *mappings*

Les données XML des sources sont stockées en utilisant les tables du schéma générique que nous avons présenté dans la sous-section précédente. Dans ce schéma, chaque nœud de données XML est stocké dans la table `XmlNode` et identifié par la colonne `nodeID`. Dans les tables stockant les *mappings*, les colonnes qui permettent de stocker les variables contiennent des références aux nœuds XML stockés dans la table `XmlNode`.

Les tables stockant les *mappings* peuvent être organisées en un **graphe de *mappings***, où chaque table est un nœud du graphe. On distingue trois types de nœuds permettant de représenter les différentes tables contenant les *mappings* :

1. Les nœuds de type ***source*** représentent les tables contenant les données extraites par les motifs. Ces nœuds sont les feuilles du graphe.
2. Les nœuds de type ***fragment*** représentent les tables contenant les données intégrées dans les fragments. Ces nœuds ont pour fils les nœuds représentant les sources utilisées pour définir le fragment.
3. Les nœuds de type ***join*** représentent les tables contenant les données intégrées dans une jointure. Ces nœuds ont pour fils les nœuds représentant les souces utilisées pour définir la jointure.

Notons que ce graphe est **acyclique**, car comme nous l'avons souligné dans le chapitre consacré à la définition du langage de vues (3.4.1), une source de données ne peut pas être définie en utilisant des données qui utilisent cette source.

Exemple Considérons le système d'intégration défini dans le chapitre précédent (Figure 4.3, page 69). Le graphe de *mappings* correspondant à ce système est représenté par la Figure 5.4.

Les motifs sur les sources sont les feuilles du graphe de *mappings*. Le nœud représentant le fragment `f_auteurs` possède deux fils qui représentent les motifs dont le fragment spécifie l'union des données. Le nœud représentant la jointure `j_livres_lirmm` possède deux fils qui représentent le fragment et le motif utilisés par la jointure.

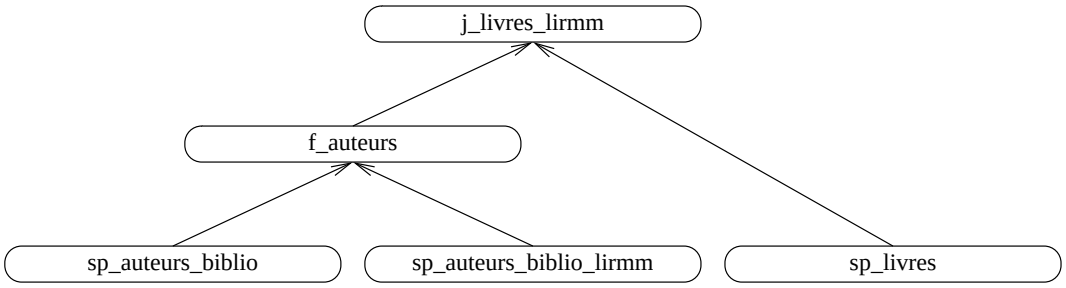

FIG. 5.4 – Graphe de *mappings* du système d'intégration.

5.2.5 Calcul des méta données

Les *mappings* spécifiés par un fragment ou une jointure peuvent être calculés avec des requêtes SQL. Ces requêtes permettent de peupler les tables représentant les fragments et les jointures d'un système d'intégration. Ces requêtes, sont définies en utilisant :

- les tables du schéma générique stockant les données XML,
- les tables représentant les sources utilisées par le fragment ou la jointure.

Représentation textuelle

Les fonctions `text(node)` et `r-text(node)` que nous avons définies dans le chapitre 3 (§ 3.2.3, page 43) peuvent s'exprimer avec SQL. Les requêtes SQL permettant d'exprimer ces fonctions seront utilisées comme des sous-requêtes pour le calcul des données d'un fragment ou d'une jointure.

```
(select value from XmlNode where nodeID = id)
union
(select value from Children c, XmlNode xn
where c.fatherID = id and xn.nodeID = c.childID
order by rank) ;
```

FIG. 5.5 – Requête SQL exprimant la fonction `text`.

La Figure 5.5 illustre la requête permettant d'exprimer la fonction `text(node)`. La notation $sql(text(id))$ permettra d'utilser cette requête comme une sous-requête. Le résultat de cette requête est une liste de valeurs textuelles (colonne `value`) qui représente la chaîne de caractères renvoyée par la fonction `text(node)`. La notation id représente l'identifiant du nœud dont on veut calculer la représentation textuelle. Le résultat de cette requête est construit en faisant l'union de deux sous-requêtes.

- La première sous-requête renvoie la colonne `value` du nœud ayant pour identifiant id. Elle permet de calculer la représentation textuelle d'un nœud de type attribut ou texte.
- La seconde sous-requête renvoie la colonne `value` des fils du nœud ayant pour identifiant id. Elle permet de calculer la représentation textuelle d'un nœud de type élément.

```
(select value from XmlNode where nodeID = id)
union
(select value from Descendant d, XmlNode xn
where d.fatherID = id and xn.nodeID = d.childID
order by rank) ;
```

FIG. 5.6 – Requête SQL exprimant la fonction `r-text`.

La Figure 5.6 illustre la requête permettant d'exprimer la fonction `r-text(node)`. La notation $sql(rtext(id))$ permettra d'utiliser cette requête comme une sous-requête. Elle fonctionne de la même façon que la requête précédente exprimant la fonction `text(node)`, mais elle utilise la table `Descendant` à la place de la table `Children`. Cela lui permet de parcourir tous les descendants du nœud id pour construire le résultat.

Calcul des données d'un fragment

Soit f un fragment intégrant les données d'une liste de sources, notée $data_f$. On note $conditions_f$ la conjonction de conditions permettant de filtrer les données du fragment et $restictions_f$ l'ensemble des restrictions permettant d'éliminer les données redondantes. Le remplissage de la table T_f contenant les données du fragment va s'effectuer en insérant successivement les données des différentes sources $s \in data_f$ utilisées par le fragment.

```
insert into T_f (id, sID, variables_s)
select getID(s, id), id, variables_s
from T_s
where sql(conditions_f) and sql(restrictions_f) ;
```

FIG. 5.7 – Requête SQL de calcul d'un fragment.

La requête SQL de la Figure 5.7 permet de réaliser les insertions dans la table contenant les données du fragment (T_f), des données d'une source s (stockées dans la table T_s). La notation $variables_s$ désigne les variables de la source s. La fonction $getID$ renvoie un identifiant pour la ligne insérée, construit à partir du nœud de la source s et de l'identifiant de la ligne dans la table de la source (`id` représentant T_s`.id`). Pour chaque insertion, seules les colonnes de T_f définies dans T_s seront instanciées, les autres recevront la valeur NULL. Cette requête SQL permet donc de construire un résultat conforme à l'opération d'union que nous avons définie dans le langage de vues (3.4.2).

La clause `where` de la requête contient les conditions qui permettent d'exprimer la composition de conditions et les restrictions du fragment. On note $conditions_f$ la composition de conditions spécifiée par le fragment f. La fonction $sql(conditions_f)$ permet de construire le code SQL exprimant ces conditions. Pour cela, on utilise les opérateurs logiques `or` et `and` définis dans SQL et éventuellement les fonctions `text` et `r-text` qui peuvent s'exprimer en SQL comme nous l'avons montré plus haut.

Pour que les données du fragment soit cohérentes avec les restrictions spécifiées, on doit exprimer ces restrictions dans la clause `where` de la requête d'insertion. L'ensemble des restrictions du fragment f est noté $restrictions_f$. La fonction $sql(restrictions_f)$ permet de construire le code SQL exprimant ces restrictions. Pour chaque restriction $r \in restrictions_f$, on ajoutera un test dans la clause `where`. Ce test vérifiera que la ligne insérée est cohérente avec r, si la source considérée est différente de celle utilisée par la restriction. La Figure 5.8 illustre la forme générale de cette condition exprimée en SQL.

`not exists`
`(select * from` $T_{prioritary_r}$
`where`
$sql(rtext(T_{prioritary_r}.variable_{1_r}))$ `=` $sql(rtext(T_s.variable_{1_r}))$ `and`
`... and`
$sql(rtext(T_{prioritary_r}.variable_{i_r}))$ `=` $sql(rtext(T_s.variable_{i_r}))$
`)`

FIG. 5.8 – Condition SQL vérifiant qu'une restriction est vérifiée.

Cette requête utilise l'opérateur `not exists` pour exprimer qu'il ne doit pas exister de lignes dans la source prioritaire ($T_{priority_r}$), qui ont la même valeur textuelle que la ligne de la source dont on insère les données (T_s). Pour cela, la clause `where` contient autant de conditions qu'il existe de variables dans la restriction. Les variables de la restriction sont notées $variable_{1_r} \ldots variable_{i_r}$. La notation $sql(rtext(T.variable_{i_r}))$ correspond à la réécriture de la fonction `r-text` pour comparer les valeurs textuelles des variables de la restriction r entre les tables de la source (T_s) et de la source prioritaire ($T_{prioritary_r}$).

Exemple Considérons le fragment `f_auteurs` (Figure 3.13, page 53) spécifiant l'union des auteurs des motifs `sp_auteurs_biblio_lirmm` et `sp_auteurs_biblio`. Les requêtes SQL permettant de peupler ce fragment sont présentées dans la Figure 5.9.

La première requête permet d'insérer dans la table représentant le fragment les données provenant de la table `sp_auteurs_biblio_lirmm`. Pour cela, elle sélectionne dans cette table toutes les lignes car c'est la table qui représente la source prioritaire.

La deuxième requête permet d'insérer dans la table représentant le fragment les données provenant de la table `sp_auteurs_biblio`. La restriction spécifiée par le fragment exprime que les données de `sp_auteurs_biblio` ne doivent pas exister dans la source prioritaire `sp_auteurs_biblio_lirmm`. Pour cela, la clause `not exists` permet d'éliminer les lignes de `sp_auteurs_biblio` dont la représentation textuelle de la variable `nom` n'existe pas dans la table représentant la source prioritaire. La représentation textuelle des variables `nom` est obtenue en utilisant la requête SQL présentée dans la Figure 5.6.

Calcul des données d'une jointure

Soit j une jointure intégrant les données de deux sources, notées $leftdata_j$ et $rightdata_j$. On note respectivement $variables_{leftdata_j}$ et $variables_{rightdata_j}$, les ensembles de variables définis sur ces sources.

```
-- insertion of data from sp_auteurs_biblio_lirmm
insert into f_auteurs (id, sID, nom, prenom, email)
select getID(sp_auteurs_biblio_lirmm, id), id, nom, prenom, email
from sp_auteurs_biblio_lirmm;

-- insertion of data from sp_auteurs_biblio
insert into f_auteurs (id, sID, nom, prenom, email)
select getID(sp_auteurs_biblio, id), id, nom, prenom, NULL
from sp_auteurs_biblio
where not exists
  (select * from sp_auteurs_biblio_lirmm
   where
     ( (select value from XmlNode where nodeID = sp_auteurs_biblio_lirmm.nom)
       union
       (select value from Descendants d, XmlNode xn
        where d.fatherID = sp_auteurs_biblio_lirmm.nom and xn.nodeID = d.childID
        order by rank) )
   = ( (select value from XmlNode where nodeID = sp_auteurs_biblio.nom)
       union
       (select value from Descendants d, XmlNode xn
        where d.fatherID = sp_auteurs_biblio.nom and xn.nodeID = d.childID
        order by rank) ) );
```

FIG. 5.9 – Requêtes permettant de peupler la table `f_auteurs`.

Les variables permettant de spécifier le prédicat de jointure sont notées respectivement $leftvariable_j$ et $rightvariable_j$. Le remplissage de la table T_j contenant les données de la jointure, va s'effectuer en executant la requête Figure 5.10.

```
insert into T_j
select getID(l.id,r.id), l.id, r.id, l.variables_leftdata_j , r.variables_rightdata_j
from T_leftdata_j l, T_rightdata_j r
where sql(rtext(l.leftvariable_j)) = sql(rtext(r.rightvariable_j)) ;
```

FIG. 5.10 – Requête SQL de calcul d'une jointure.

Cette requête effectue la jointure des tables $T_{leftdata_j}$ et $T_{rightdata_j}$ représentant les données des sources à intégrer. Le prédicat de jointure s'exprime en utilisant la fonction `r-text`. De cette façon, la jointure s'effectue sur la représentation textuelle des nœuds référencés par les variables du prédicat.

Exemple Considérons la jointure `j_livres_lirmm` (Figure 3.13, page 53) spécifiant le croisement des informations concernant des auteurs et des livres. La requête SQL permettant de peupler cette jointure est présentée dans la Figure 5.11.

```
insert into j_livres_lirmm
select getID(l.id, r.id), l.id, r.id,
       l.titre, l.auteur, l.prix, r.nom, r.prenom, r.email
from sp_livres_lirmm l, f_auteurs r
where
  ( (select value from XmlNode where nodeID = l.auteur)
    union
    (select value from Descendants d, XmlNode xn
     where d.fatherID = l.auteur and xn.nodeID = d.childID
     order by rank) )
= ( (select value from XmlNode where nodeID = r.nom)
    union
    (select value from Descendants d, XmlNode xn
     where d.fatherID = r.nom and xn.nodeID = d.childID
     order by rank) );
```

FIG. 5.11 – Requête permettant de peupler la table `j_livres_lirmm`.

La requête permet d'insérer dans la table représentant la jointure les données provenant des tables `sp_livres_lirmm` et `f_auteurs`. Pour cela, le prédicat de jointure utilise les représentations textuelles des variables `auteur` et `nom`.

5.3 Maintenance de vues VIMIX

Dans les documents XML, la structure et les données sont mélangées. Cependant, on peut distinguer deux sortes d'évolution : l'**évolution structurelle** et l'**évolution des données**. Dans le contexte relationnel, l'évolution de la structure des données (c'est à dire du schéma relationnel) est distinguée clairement de celle des données. Ce n'est pas le cas avec XML, car la structure est contenue dans les données. Cependant, on peut illustrer ces deux types d'évolution en considérant des données qui sont validées par une DTD. Lorsque la DTD est modifiée, la structure des données va évoluer pour rester cohérente avec la nouvelle DTD. Dans ce cas on parle d'évolution structurelle, sinon on parle d'évolution des données. On parlera ici d'évolution structurelle pour une source de données lorsque la spécification des vues utilisant cette source doit être modifiée pour prendre en compte cette évolution. Nous avons considéré les deux problèmes suivants :

1. le rafraîchissement des vues lorsque les données des sources évoluent,
2. l'évolution des vues lorsque la structure des sources de données évolue.

Nous allons montrer ici que notre architecture de stockage, grâce au mécanisme d'identifiant proposé, permet de traiter de manière incrémentale le **rafraîchissement** et l'**évolution** des vues.

Les vues VIMIX permettent d'intégrer des données XML provenant de sources multiples et hétérogènes. Cela offre la possibilité d'utiliser les sources de données disponibles sur le *Web* ou produites par diverses applications. Dans ce contexte, nous pensons que ces sources peuvent difficilement être monitorées. La plus petite opération de rafraîchissement de l'entrepôt est donc le rafraîchissement d'une source.

5.3.1 Rafraîchissement des vues

Nous avons traité le rafraîchissement des vues pour des sources qui ne sont pas monitorées. La granularité minimale de mise à jour sera donc le rafraîchissement d'une source de données. Notre méthode est incrémentale car elle ne nécessite pas de recalculer toutes les données stockées. Pour cela, nous utilisons les identifiants des *mappings* des motifs, des fragments et des jointures.

Rafraîchissement des données d'une source

L'algorithme 1 présente la stratégie générale de rafraîchissement des sources de données. La fonction `refresh-source` accepte en paramètre la source de données que l'on souhaite rafraîchir, notée s.

Algorithme 1: `refresh-source(`s`)`

Résultat : Rafraîchissement des données de la source s
pour chaque *motif sp utilisant la source s* **faire**
  copier la table T_{sp} dans $T_{deletes}$;
  vider la table T_{sp} ;
  extraire les données provenant de la source s pour peupler T_{sp} ;
  pour chaque T_{father} *père de* T_{sp} *dans le graphe de mappings* **faire**
    `refresh-mappings(`T_{father}`, `sp`, `$T_{deletes}$`, `T_{sp}`)` ;
supprimer les données provenant de la source de s ;

Cet algorithme fonctionne de la manière suivante. Pour chaque motif sp défini sur cette source, on copie le contenu de la table T_{sp} dans $T_{deletes}$ et on vide la table T_{sp}. On répète le processus d'extraction des données sur cette source, pour peupler à nouveau la table T_{sp}.

A ce stade, les données d'un motif sur la source à rafraîchir sont mises à jour. Il faut ensuite mettre à jour les *mappings* des fragments et des jointures qui utilisent ces données. Pour cela, on execute la fonction `refresh-mappings` pour toutes les tables qui sont des nœuds pères de T_{sp} dans le graphe de *mappings*.

Enfin, on supprime les données XML provenant de la source s qui sont stockées dans les tables du schéma générique. Les suppressions ne peuvent pas être faites plus tôt, car les données des anciens *mappings* peuvent être nécessaire pour la maintenance.

Rafraîchissement des fragments et des jointures

L'algorithme 2 présente la stratégie de mise à jour d'une table représentant un fragment ou une jointure dans le graphe de *mappings*. La fonction `refresh-mappings` accepte en argument quatre paramètres :

1. La table T_s contenant les données d'un fragment ou d'une jointure qui doit être mise à jour.
2. La source *child* représentant le motif, le fragment ou la jointure qui a été mis à jour.
3. La table $T_{deletes_{child}}$ contenant les lignes qui ont été supprimées dans la table T_{child}.
4. La table $T_{inserts_{child}}$ contenant les lignes qui ont été ajoutées dans la table T_{child}.

Algorithme 2: `refresh-mappings(`T_s`,` $child$`,` $T_{deletes_{child}}$`,` $T_{inserts_{child}}$`)`

Résultat : Rafraîchissement des *mappings* d'un fragment ou d'une jointure stockés dans T_s
calculer dans $T_{deletes}$ les lignes à supprimer dans T_s (en utilisant $T_{deletes_{child}}$) ;
calculer dans $T_{inserts}$ les lignes à ajouter dans T_s (en utilisant $T_{inserts_{child}}$) ;
supprimer dans T_s les lignes de $T_{deletes}$;
ajouter dans T_s les lignes de $T_{inserts}$;
pour chaque T_{father} *père de* T_s *dans le graphe de mappings* **faire**
  `refresh-mappings(`T_{father}`,` s`,` $T_{deletes}$`,` $T_{inserts}$`)` ;

Cet algorithme est incrémental, car il utilise les données supprimées et ajoutées dans la source mise à jour pour effectuer seulement les modifications nécessaires. Il peut se décomposer en trois étapes.

1. Tout d'abord, on calcule les lignes du fragment ou de la jointure à supprimer et ajouter. Ce calcul utilise les lignes supprimées et ajoutées dans la source qui est à l'origine de la mise à jour. Les requêtes SQL permettant d'exprimer ces calculs, dépendent du type de la table à mettre jour (fragment ou jointure). Elles sont présentées en annexe (D, page 161).
2. Ensuite, les lignes à supprimer (dans la table $T_{deletes}$) et les lignes à ajouter (dans la table $T_{inserts}$), sont respectivement supprimées et ajoutées de la table à rafraîchir (T_s).
3. Enfin, les mises à jour sont propagées aux tables pères du graphe de *mappings*. Cette propagation est réalisée par un appel récursif de la fonction. La condition est réalisée par les tables qui n'ont pas de père, ce qui est assurée par le fait que le graphe de *mappings* est acyclique.

Propagation des rafraîchissements

La Figure 5.12 illustre le fonctionnement des algorithmes que nous avons proposé pour le rafraîchissement des données. Pour des raisons d'espace, nous n'avons pas utilisé les vues définies

précédemment pour illustrer le fonctionnement de ces algorithmes. Le graphe de *mappings* utilisé sur cet exemple est constitué de trois motifs sur des sources, notés $sp1$, $sp2$ et $sp3$. Le fragment f contient l'union des données des motifs de $sp1$ et $ssp2$. Enfin, j contient la jointure des données de f et de $sp3$, en utilisant les variables a et c pour définir le prédicat de jointure. Pour faciliter la lisibilité de l'exemple, les motifs stockent les valeurs des éléments XML correspondants à l'instanciation des variables, plutôt que les références à ces éléments (qui devraient être stockés en utilisant le schéma générique).

Les instances des tables sont représentées sous forme tabulaire. Pour les tables qui seront rafraîchies par la propagation des mises à jour, on présente les deux instances. L'instant `(0)` représente l'état initial des données. L'instant `(i)` représente l'état lors de la i^{e} propagation.

Les données ajoutées et supprimées (notées $T_{inserts}$ et $T_{deletes}$ dans les algorithmes), sont encadrées par des pointillés. Les données encadrées dans l'instance d'une table à l'état initial sont des données à supprimer, sinon ce sont des données à ajouter.

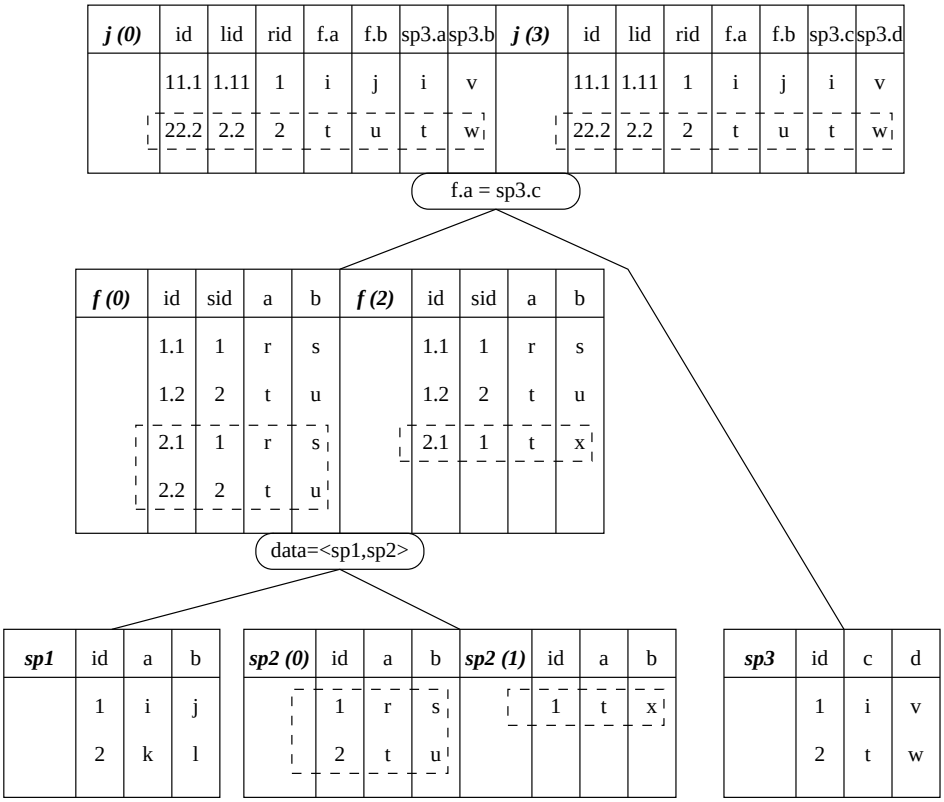

FIG. 5.12 – Illustration de la propagation des rafraîchissements.

La Figure 5.12 illustre la propagation des rafraîchissements lorsque la source du motif de $sp2$ est modifiée. La première étape du processus de rafraîchissement est de vider la table T_{sp1}, puis d'extraire les données de la source de sp. Ensuite, la deuxième étape propage le rafraîchissement à la table T_f, car elle utilise T_{sp2}. Les données qui avaient supprimées lors de la première étape, permettent de calculer

les données à supprimer dans T_f. Ensuite, on ajoute les données qui avaient été ajoutées lors de la première étape, puis on propage le rafraîchissement à la table T_j car elle utilise T_f. Cette troisième étape se déroule de la même manière que les précédentes, en calculant les lignes à ajouter puis celles à supprimer.

5.3.2 Evolution des vues

Notre architecture de stockage permet de traiter de façon incrémentale l'évolution des vues. En effet, il n'est pas nécessaire de recalculer toutes les données lorsque la spécification d'une vue, d'un motif, d'un fragment ou d'une jointure évolue.

Avec notre méthode de stockage, les données permettant de construire le résultat d'une vue sont stockées dans une table contenant les données extraites. Le document XML représentant le résultat d'une vue n'est pas stocké. Si on modifie la spécification d'une vue, il n'y a donc pas de modifications à propager dans l'entrepôt.

En effet, si on modifie la source de données d'une vue, il n'y a pas de modification à propager car la vue doit utiliser une source de données qui est déjà définie dans l'entrepôt. On peut aussi faire évoluer la spécification d'une vue en modifiant le motif décrivant la forme de son résultat. Là encore, il n'y a pas de modification à propager car le résultat de la vue est construit dynamiquement, lorsque c'est nécessaire. De la même façon, on peut ajouter de nouvelles vues dans l'entrepôt sans effet sur les données stockées.

Evolution de la spécification des motifs sur les sources

La fonction `evolution-spattern` présentée dans l'algorithme 3 permet de maintenir les données stockées, lorsque la spécification d'un motif sur une source évolue. Cette évolution nécessite de répéter le processus d'extraction des données du motif. La nouvelle spécification peut modifier l'ensemble de variables qui est défini par le motif. Si l'ensemble des variables est modifié, il faut modifier le schéma de la table représentant le motif (T_{sp}) afin qu'il soit cohérent avec la nouvelle spécification du motif.

Si le schéma de la table représentant le motif (T_{sp}) n'est pas modifié, alors on peut traiter l'évolution de la spécification comme un simple rafraîchissement des données. En effet, l'évolution de la spécification du motif sera transparente pour les fragments et les jointures qui l'utilisent. On peut donc propager l'évolution avec la fonction `refresh-mappings` (algorithme 2) qui a été définie pour le rafraîchissement des données. Comme pour la mise à jour des données, on propage la modification à toutes les tables utilisant T_{sp} dans le graphe de *mappings*.

Algorithme 3: `evolution-spattern(`sp`)`

Résultat : Evolution de l'entrepôt lorsque le motif sp est modifié
copier la table T_{sp} dans $T_{deletes}$;
vider la table T_{sp} ;
modifier le schéma de T_{sp} pour qu'il soit conforme à $variables_{sp}$;
extraire les données de la source de sp pour peupler T_{sp} ;
pour chaque T_{father} *père de* T_{sp} *dans le graphe de mappings* **faire**
  si *le schéma de* T_{sp} *a été modifié* **alors**
    `evolution-mappings(`T_{father}`,` sp`,` $T_{deletes}$`,` T_{sp}`)` ;
  sinon
    `refresh-mappings(`T_{father}`,` sp`,` $T_{deletes}$`,` T_{sp}`)` ;

Si l'évolution de la spécification a entraîné une modification du schéma de la table représentant le motif, il est quand même possible de propager de manière incrémentale cette évolution. Pour cela, on utilise la fonction `evolution-mappings` qui est présentée dans l'algorithme 4. L'évolution est propagée à toutes les tables utilisant T_{sp} dans le graphe de *mappings*.

Algorithme 4: `evolution-mappings(`T_s`,` $child$`,` $T_{deletes_{child}}$`,` $T_{inserts_{child}}$`)`

Résultat : Evolution des tables contenant des *mappings*
calculer dans $T_{deletes}$ les lignes à supprimer dans T_s (en utilisant $T_{deletes_{child}}$) ;
supprimer dans T_s les lignes de $T_{deletes}$;
modifier le schéma de T_s pour qu'il soit conforme à $variables_s$;
calculer dans $T_{inserts}$ les lignes à ajouter dans T_s (en utilisant $T_{inserts_{child}}$) ;
ajouter dans T_s les lignes de $T_{inserts}$;
pour chaque T_{father} *père de* T_s *dans le graphe de mappings* **faire**
  si *le schéma de* T_s *a été modifié* **alors**
    `refresh-mappings(`T_{father}`,` s`,` $T_{deletes}$`,` $T_{inserts}$`)` ;
  sinon
    `evolution-mappings(`T_{father}`,` s`,` $T_{deletes}$`,` $T_{inserts}$`)` ;

Lorsqu'on propage l'évolution de la spécification d'une source *child* à une source s, c'est que le schéma de table représentant *child* a été modifié. Ces modifications peuvent éventuellement avoir une incidence sur les variables définies par s. Si c'est le cas, il faut modifier le schéma de T_s pour qu'il soit cohérent avec $variables_s$.

L'évolution de la spécification de la source *child*, signifie que ses données ont également été mises à jour. Il faut donc mettre à jour les données de la source s, pour qu'elles soient cohérentes avec les

données de sa source qui a été modifié. On peut calculer les données à supprimer et à ajouter dans T_s avec des requêtes SQL. Ces requêtes sont similaires à celles utilisées pour calculer les données qui doivent être supprimées et ajoutées lors du rafraîchissement d'une source.

Enfin, il faut propager l'évolution de la source s qui a été mise à jour. Si l'évolution de cette source n'a pas entraînée de modification du schéma de la table T_s, alors cette évolution sera transparente pour les sources utilisant s. Il suffit donc de mettre à jour les données des tables utilisant T_s dans le graphe de *mappings*, en utilisant la fonction `refresh-mappings`.

Dans le cas contraire, l'évolution du schéma de T_s peut avoir une incidence sur le schéma des tables qui l'utilisent dans le graphe de *mappings*. Il faut alors propager l'évolution de T_s en utilisant la fonction `evolution-mappings`.

Remarquons que cette fonction peut s'appliquer à des tables représentant des fragments ou des jointures. En fonction du type de la source qui évolue, les requêtes SQL qui permettent de calculer les données à supprimer et à ajouter auront une forme différente, comme pour le rafraîchissement des données.

Cette fonction `evolution-mappings` est également utilisée, lorsque la spécification d'un fragment ou d'une jointure évolue.

Evolution de la spécification des fragments

La fonction `evolution-fragment` présentée par l'algorithme 5 permet de maintenir les données de l'entrepôt, lorsque la spécification d'un fragment évolue. Cette évolution peut concerner deux types de propriétés : la liste de ses sources de données ou les conditions et restrictions permettant de filtrer les données. Pour connaître les propriétés qui ont été modifiées, on passe en paramètre à la fonction `evolution-fragment` l'ancienne spécification du fragment (f_{old}).

Si la source de données du fragment a été modifiée, il faut recalculer toutes les données contenues dans la table qui représente le fragment (T_f). En effet, l'ordre et le nombre des sources de données sont des paramètres déterminant pour calculer les valeurs des identifiants des *mappings* du fragment. La modification de la liste de la source rend donc obsolète les identifiants qui permettent de maintenir à jour de façon incrémentale les données. Il faut recalculer toutes les données de cette table, ce qui permettra de générer de nouveaux identifiants.

De plus, la modification de la source de données peut également avoir des conséquences sur la liste des variables définies par le fragment. Par exemple, si on ajoute une source de données possédant une variable qui n'appartient pas à $variables_{f_{old}}$, il faut mettre à jour le schéma de la table T_f pour qu'il soit cohérent avec $variables_f$.

Enfin, pour chaque table utilisant T_f, on propage incrémentalement la modification avec `evolution-mapping` si le schéma de T_f a été modifié et avec `refresh-mappings` sinon.

Algorithme 5: `evolution-fragment(`f`,` f_{old}`)`

Résultat : Evolution d'un fragment f lorsque sa spécification est modifiée
si *la liste des sources de données a été modifiée* **alors**
  copier la table T_f dans $T_{deletes}$;
  vider la table T_f ;
  modifier si nécessaire le schéma de T_f pour qu'il soit conforme à $variables_f$;
  calculer les mappings de f pour peupler T_f ;
  pour chaque T_{father} *père de* T_f *dans le graphe de mappings* **faire**
    si *le schéma de* T_f *a été modifié* **alors**
      `evolution-mappings(`T_{father}`,` f`,` $T_{deletes}$`,` T_f`)` ;
    sinon
      `refresh-mappings(`T_{father}`,` f`,` $T_{deletes}$`,` T_f`)` ;
sinon
  calculer dans $T_{deletes}$ les lignes à supprimer dans T_f ;
  calculer dans $T_{inserts}$ les lignes à ajouter dans T_f ;
  supprimer dans T_f les lignes de $T_{deletes}$;
  ajouter dans T_f les lignes de $T_{inserts}$;
  pour chaque T_{father} *père de* T_f *dans le graphe de mappings* **faire**
    `refresh-mappings(`T_{father}`,` f`,` $T_{deletes}$`,` $T_{inserts}$`)` ;

Si la spécification des conditions ou des restrictions sont les seules évolutions de la spécification de f, alors on peut calculer les données à supprimer et à ajouter dans T_f. En effet, l'évolution de la spécification de f ne signifie pas que les données de ces sources ont été modifiées. La requête SQL permettant de calculer les données à supprimer dans T_f, devra rechercher dans T_f les lignes qui ne respectent pas les nouvelles conditions (ou restrictions). La requête SQL permettant de calculer les données à ajouter dans T_f, devra rechercher dans les tables représentant les sources de f les lignes qui respectent les conditions (ou les restrictions) qui ont été supprimées.

Ces modifications n'ont pas d'incidence sur le schéma de T_f, on peut donc propager l'évolution comme un rafraîchissement des données, en utilisant la fonction `refresh-mappings`.

Evolution de la spécification des jointures

La fonction `evolution-join` présentée par l'algorithme 6 permet de maintenir les données de l'entrepôt, lorsque la spécification d'une jointure évolue. Cette évolution peut concerner deux types de propriétés : une source de données ou une variable constituant le prédicat de la jointure. Dans les deux cas, il faut recalculer entièrement le résultat de la jointure. C'est pour cette raison, qu'il n'est pas nécessaire de passer en paramètre l'ancienne spécification de la jointure à la fonction `evolution-join`.

Algorithme 6: `evolution-join(`j`)`

Résultat : Evolution d'une jointure j lorsque sa spécification est modifiée
copier la table T_j dans $T_{deletes}$;
vider la table T_j ;
modifier le schéma de T_j pour qu'il soit conforme à $variables_j$;
calculer les *mappings* de j pour peupler T_j ;
pour chaque T_{father} *père de* T_j *dans le graphe de mappings* **faire**
  si *le schéma de* T_j *a été modifié* **alors**
    `evolution-mappings(`T_{father}`,` j`,` $T_{deletes}$`,` T_j`)` ;
  sinon
    `refresh-mappings(`T_{father}`,` sp`,` $T_{deletes}$`,` T_{sp}`)` ;

La modification d'une source de la jointure peut entraîner la modification du schéma de la jointure. Par exemple, si la nouvelle source de la jointure contient des variables qui n'étaient pas présentes dans l'ancienne source, il faut ajouter les colonnes correspondantes dans la table représentant la jointure (T_j). Si le schéma de T_j a été modifié, alors on propage l'évolution avec la fonction `evolution-mappings`. Dans le cas contraire, l'évolution de la spécification de la jointure peut être traitée comme un rafraîchissement des données, et propagée avec `refresh-mappings`.

Ajouts et suppressions de sources dans l'entrepôt

L'ajout et la suppression de sources de données peuvent être traités avec les fonctions que nous avons présentées. Lorsqu'on supprime une source de données, il faut supprimer tous les motifs qui ont été définis sur cette source. Ces suppressions entraînent l'évolution de la spécification de chaque fragment et de chaque jointure qui utilisaient ces motifs. Les fonctions que nous avons définies permettent de propager incrémentalement ces évolutions.

L'ajout d'une nouvelle source de données nécessite l'ajout d'au moins un motif sur cette source. Le processus d'extraction des données sur cette source permettra de créer les tables qui représentent les nouveaux motifs. Ces tables seront ajoutées dans le graphe de *mappings*. Si le motif est utilisé par un fragment ou une jointure, il entraînera l'évolution de sa spécification. Cette évolution sera alors propagée incrémentalement avec les fonctions que nous définies.

L'intégration de données avec VIMIX permet de simuler le fonctionnement d'une approche *LAV*. Les motifs sur les sources peuvent être considérés comme des vues des sources locales sur le schéma médiateur. Lorsqu'un motif sur une nouvelle source de données est ajouté à un fragment, les variables définies par le fragment ne sont pas nécessairement modifiées. Dans ce cas, le schéma médiateur ne sera pas modifié et l'ajout de la nouvelle source sera traité comme le rafraîchissement des données.

Cette caractéristique permet de simuler le fonctionnement d'une approche *LAV* lorsqu'on modifie ou qu'on ajoute de nouvelles sources de données.

5.4 Construction du résultat de vues VIMIX

Nous utilisons un SGBD relationnel pour stocker les données XML des sources et les *mappings* des vues. La définition d'une vue contient un arbre qui spécifie la structure et les données de son résultat. Cet arbre est utilisé pour construire le graphe de données XML qui contient le résultat de la vue. Nous allons tout d'abord présenter les fonctions qui permettent de construire un graphe de données XML.

5.4.1 Construction d'un graphe de données XML

Pour construire de nouveaux nœuds de données XML, on dispose des fonctions qui sont présentées dans le Tableau 5.3. Ces fonctions permettent de construire de nouveaux nœuds de type élément, attribut ou texte. Elles sont utilisées pour construire les graphes de données que nous avons présenté au chapitre 3, dans la section consacrée au modèle de données (§ 3.2, page 38).

Fonction	**Description**
`new-document(`*rootname*`)`	Construit un nœud de type élément qui sera la racine d'un nouveau document, en spécifiant son nom (*name*).
`new-element(`*name*`, `*father*`)`	Construit un nœud de type élément, en spécifiant son nom (*name*) et son père (*father*).
`new-attribute(`*name*`, `*father*`)`	Construit un nœud de type attribut qui n'a pas de valeur, en spécifiant son nom (*name*) et son père (*father*).
`new-text(`*value*`, `*father*`)`	Construit un nœud de type texte, en spécifiant sa valeur (*value*) et son père (*father*).
`new-attribute(`*name*`, `*value*`, `*father*`)`	Construit un nœud de type attribut, en spécifiant son nom (*name*), sa valeur (*value*) et son père (*father*).

TAB. 5.3 – Fonctions de onstruction de nœuds XML.

La racine du graphe de données est construite grâce à la fonction `new-document`. Les liens du graphe sont construits grâce au paramètre *father* qui permet de spécifier le nœud père du nœud à construire. Les nœuds de type texte ne peuvent pas être utilisés comme nœuds pères, car ils n'acceptent pas de fils

dans le modèle de données que nous avons présenté. De même, les nœuds de type attribut ne peuvent pas être utilisés comme pères pour la construction d'un autre attribut.

Le type de lien construit par une fonction dépend du type du nœud père passé en paramètre et du type du nœud construit.

- Lorsque le nœud père est de type élément, un lien de composition est créé entre ce nœud et le nœud construit.
- Lorsque le nœud père est de type attribut, le type du lien construit dépend du type du nœud renvoyé par la fonction :
  - si c'est un nœud de type texte (correspondant à la valeur de l'attribut), la fonction construit un lien de composition,
  - si c'est un nœud de type élément, la fonction construit un lien de référence.

Pour construire des liens, on dispose aussi d'une fonction `new-link(`$father$`,` $child$`)`. Cette fonction permet de créer un lien entre le nœud père ($father$) et le nœud fils ($child$). Le type du lien créé par cette fonction dépend des types du nœud père et du nœud fils. Cette fonction est utile pour construire un graphe de données, car elle permet de construire un lien sans construire un nouveau nœud. Cela est nécessaire lorsqu'il existe des liens de référence dans un graphe. En effet, les nœuds éléments référencés dans un graphe de données sont aussi imbriqués dans d'autres éléments. Il est alors nécessaire de ne pas dupliquer ces nœuds pour construire le graphe de données.

La fonction `new-attribute(`$name$`,` $value$`,` $father$`)` ne fait pas partie du jeu minimal de fonctions pour construire un graphe de données XML. Nous l'avons proposé pour faciliter l'écriture de la construction du graphe de données. En effet, cette fonction peut être réécrite par l'utilisation d'autres fonctions. En fait, la valeur d'un attribut est un nœud de type texte qui est fils de l'attribut. On a donc des constructions équivalentes (mais les fonctions ne renvoient pas le même type de nœud) avec :

`new-attribute(`$name$`,` $value$`,` $father$`)` $\Longleftrightarrow$ `new-text(`$value$`, new-attribute(`$name$`,` $father$`))`

5.4.2 Recomposition des données XML des sources

Les données XML des sources sont stockées dans un SGBD relationnel. Pour insérer ces données dans le résultat d'une vue, une phase de recomposition est nécessaire. Pour interroger les tables du schéma générique, on dispose d'une fonction `query` qui permet d'exécuter une requête SQL passée en paramètre. Une requête SQL renvoie toujours une table, qui peut être vue comme une liste de lignes. Pour alléger l'écriture des algorithmes, nous considérons que le type du résultat de la fonction `query` dépend du résultat de la requête q passée en paramètre. Si le résultat de q ne contient qu'une seule ligne, alors `query` renverra une ligne. Lorsqu'une ligne ne contient qu'un seul élément, alors la fonction renverra une liste de cet élément. Par exemple, la requête `'select childID from Children'` ne renverra pas une liste de lignes mais une liste de `childID`.

La fonction `node-reconstruction` présentée dans l'algorithme 7 permet de reconstruire un nœud XML, à partir des données stockées dans les tables du schéma générique. Elle accepte les paramètres

suivants.

- $nodeID$ est l'identifiant du nœud dans la table `XmlNode`.
- $father$ est le nœud du graphe de données XML qui est le père du nœud à construire.

Algorithme 7: `node-reconstruction(`$nodeID$`,` $father$`)`

Résultat : Reconstruction d'un nœud de données XML à partir des tables du schéma générique
si $nodeID \in G_{father}.N$ **alors**
  $child \leftarrow$ `get-node(`$nodeID$`,` $G_{father}.N$`)` ;
  `new-link(`$father$`,` $child$`)` ;
sinon
  $line \leftarrow$ `query('select * from XmlNode where nodeID =` $nodeID$`')` ;
  si $line.elemID = \text{NULL} \wedge line.attID = \text{NULL}$ **alors**
    `new-text(`$line.value$`,` $father$`)` ;
  sinon
    si $line.elemID \neq \text{NULL}$ **alors**
      $name \leftarrow$ `query('select * from Element where elemID =` $line.elemID$`')` ;
      $newnode \leftarrow$ `new-element(`$name$`,` $father$`)` ;
    sinon
      $name \leftarrow$ `query('select * from Attribute where attID =` $line.attID$`')` ;
      $newnode \leftarrow$ `new-attribute(`$name$`,` $line.value$`,` $father$`)` ;
    $childrenID \leftarrow$ `query('select childID from Children where fatherID =` $nodeID$
    `order by rank')` ;
    pour chaque $childID \in childrenID$ **faire**
      `node-reconstruction(`$childID$`,` $newnode$`)` ;

Un nœud du graphe de données peut avoir plusieurs pères, grâce aux liens de références. Lorsqu'on veut reconstruire un nœud du graphe de données, ce nœud peut avoir déjà été construit par un autre nœud père. Si c'est le cas, il suffit d'ajouter un lien entre le nœud père ($father$) et le nœud qui est déjà construit ($child$). La fonction `new-link` détermine le type du lien à construire (composition ou référence) en fonction des types des nœuds passés en paramètres.

Si le nœud n'existe pas déjà dans le graphe, alors il faut le construire en utilisant les fonctions que nous avons présentées dans le Tableau 5.3. Pour cela, on cherche dans la table `XmlNode` la ligne représentant le nœud à reconstruire ($line$). Le reste du traitement dépend du type du nœud à reconstruire.

Si c'est un nœud de type texte ($line.elemID = \text{NULL} \wedge line.attID = \text{NULL}$), sa valeur est stockée dans la ligne de la table `XmlNode` ($line.value$). On construit alors le nouveau nœud avec la fonction `new-text`.

Si c'est un nœud de type élément ($line.elemID \neq$ NULL), on recherche le nom de cet élément dans la table `Element`. On construit alors le nouveau nœud avec la fonction `new-element`. Si le nœud à reconstruire est de type attribut (il n'est ni de type texte, ni de type élément), on recherche le nom de cet attribut dans la table `Attribute`. La valeur de cet attribut est stockée dans la ligne de la table `XmlNode` ($line.value$). On construit alors le nouveau nœud avec la fonction `new-attribute`.

Pour les nœuds de type élément ou attribut, il faut aussi construire tous les nœuds fils. Pour cela, on cherche ces fils dans la table `Children` et on appelle récursivement la fonction `node-reconstruction`. Le nœud père utilisé lors de cet appel, est le nœud qui vient d'être construit ($newnode$). Cela permet de reconstruire récursivement la hiérarchie des nœuds dans le graphe.

5.4.3 Construction des données d'une vue

Pour chaque vue, on stocke sa définition et une table contenant les *mappings* qui permettent de construire son résultat. La fonction `view-construction` présentée dans l'algorithme 8 permet de reconstruire le résultat d'une vue. Elle accepte les paramètres suivants.

- $rtree_v$ est l'arbre qui spécifie le résultat de la vue.
- T est la table qui contient les *mappings* des variables de la vue. Ces *mappings* permettent d'associer les variables utilisées dans $rtree_v$ aux données stockées dans les tables du schéma générique.
- gb_v est la liste des variables qui indique les niveaux de regroupement du résultat de la vue.
- $father$ est le nœud du graphe de données XML qui sera le père du résultat de la vue.

Si le résultat de la vue ne spécifie pas de niveaux de regroupement ($gb_v =<>$), la construction du résultat fonctionne de la manière suivante. Pour chaque ligne de la table contenant les *mappings* de la vue, on instancie l'arbre qui spécifie le résultat de la vue, avec la fonction `instanciate`, présentée plus bas (§ 5.4.4).

Si la spécification de la vue utilise des niveaux de regroupement, alors la construction du résultat s'effectue en plusieurs étapes. On instancie la branche du résultat qui contient la variable spécifiant le premier niveau de regroupement (gb_{v_1}) et on construit le reste du résultat en appelant récursivement la fonction `view-reconstruction`.

La fonction `get-branch` permet d'extraire la branche de l'arbre du résultat contenant la variable gb_{v_1}. La fonction `get-subtrees` permet d'extraire les sous-arbres dont les racines sont des nœuds fils des nœuds d'une branche de l'arbre. La Figure 5.13 illustre le fonctionnement de ces fonctions.

Considérons la vue `v_livres_lirmm` (Figure 3.19, page 62). L'arbre spécifiant le résultat de la vue est composé de nœuds de type `element`, `attribute` et `expression`. La branche de cet arbre contenant la variable `auteur` est entourée par un trait plein. Les sous-arbres dont la racine est un élément fils de cette branche sont entourés par un trait en pointillé.

Algorithme 8: `view-construction(`$rtree_v$`,` gb_v`,` T`,` $father$`)`

Résultat : Construction des données XML du résultat d'une vue
si gb_v =<> **alors**
  $lines \leftarrow$ `query('select * from` T`')` ;
  pour chaque $line \in lines$ **faire**
    `instanciate(`$root_{tree_v}$`,` $line$`,` T`,` $father$`)` ;
sinon
  $branch \leftarrow$ `get-branch(`$rtree_v$`,` gb_{v_1}`)` ;
  $subtrees \leftarrow$ `get-subtrees(`$rtree_v$`,` $branch$`)` ;
  $T_{text} \leftarrow$ table contenant la représentation textuelle de gb_{v_1} ;
  pour chaque $line \in$ `query('select distinct` gb_{v_1} `from` T_{text}`)` **faire**
    $value \leftarrow line.gb_{v_1}$;
    `instanciate(`$branch$`,` $line$`,` T`,` $father$`)` ;
    pour chaque $subtree \in subtrees$ **faire**
      $gb_{temp} \leftarrow gb_v \cap variables_{subtree}$;
      $T_{temp} \leftarrow$ `query('select distinct` $variables_{subtree}$ `from` T `where` gb_{v_1} `=` $value$`')` ;
      $father_{temp} \leftarrow$ nœud de G_{father} qui a été instancié pour le nœud de $branch$ qui est père de $subtree$;
      `view-construction(`$subtree$`,` gb_{temp}`,` T_{temp}`,` $father$`)` ;

Après avoir construit la branche et les sous-arbres qui correspondent à la variable définissant le niveau de regroupement, on construit une table T_{text} dans laquelle on ajoute la représentation textuelle de cette variable. Pour chaque ligne de cette table, on instancie la branche et les sous-arbres précédemment calculés. L'instanciation des sous-arbres se fait en appelant récursivement la fonction `view-reconstruction`. Les paramètres utilisés pour cet appel sont les suivants.

- $subtree$ est le sous-arbre contenant la spécification de la partie de la vue à reconstruire.
- gb_{temp} est la liste des variables indiquant les niveaux de regroupement dans la partie de la vue à reconstruire.
- T_{temp} est une table contenant les *mappings* pour les variables de la partie de la vue à reconstruire. Cette table est calculée en projetant les variables qui sont utilisées par le sous-arbre ($variables_{subtree}$ et en sélectionnant les lignes de T_{text} qui correspondent au regroupement (gb_{v_1} `= '`$value$`'`).
- $father_{temp}$ est le nœud du graphe de données qui est le père de la partie de la vue à reconstruire. Ce nœud est celui qui a été instancié pour le nœud de $branch$ qui est le père du sous-arbre considéré ($subtree$).

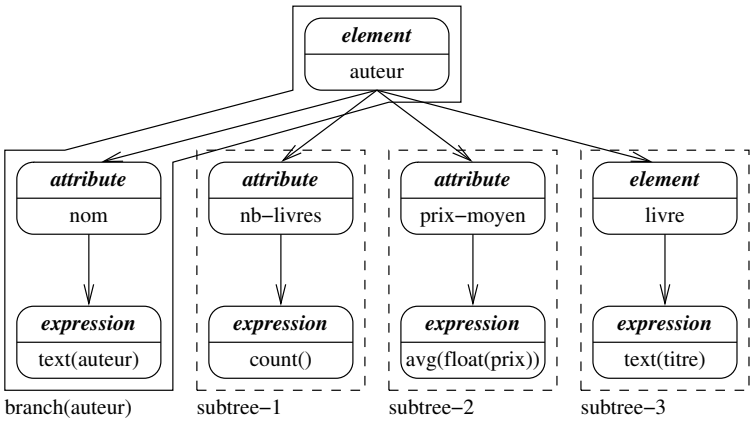

FIG. 5.13 – Fonctions `get-branch` et `get-subtrees`.

5.4.4 Instanciation des données XML d'une vue

La fonction `instanciate` présentée dans l'algorithme 9 permet d'instancier l'arbre (ou le sous-arbre) du résultat d'une vue. Elle accepte les paramètres suivants.

- *rnode* est le nœud de l'arbre (ou du sous-arbre) du résultat à instancier.
- *line* est la ligne contenant les *mappings* des variables à instancier.
- P est la partition contenant les lignes utilisées pour appliquer les fonctions d'agrégation.
- $father$ est le nœud du graphe de données XML qui sera le père du nœud à instancier.

Algorithme 9: `instanciate`(*rnode*, *line*, P, $father$)

Résultat : Instancie les données XML d'une partie du résultat d'une vue
si *rnode.type* = `element` **alors**
  newnode ← `new-element`(*rnode.value*, $father$) ;
  pour chaque *child* ∈ *rnode.children* **faire**
    `instanciate`(*child*, *line*, *newnode*) ;
sinon si *rnode.type* = `attribute` **alors**
  newnode ← `new-attribute`(*rnode.value*, $father$) ;
  `instanciate-expression`(*line*.(*rnode.child.value*), P, *newnode*) ;
sinon
  `instanciate-expression`(*line*.(*rnode.value*), P, $father$) ;

Cette fonction parcourt l'arbre (ou le sous-arbre) du résultat de la vue à instancier. Si le nœud du résultat (*rnode*) est de type élément, alors on crée un nouvel élément avec la fonction `new-element`. Ensuite, l'appel récursif à la fonction permet d'instancier chaque fils (*child*) de *rnode* dans l'arbre du résultat. Le nœud du graphe de données qui sera le père de ces instanciations est le nœud qui vient d'être construit (*newnode*).

Si le nœud du résultat est de type attribut, alors on crée un nouvel attribut avec la fonction `new-attribute`. La valeur de cet attribut est spécifiée par la variable du nœud fils du résultat (*rnode.child.value*). On construit la valeur de l'attribut avec la fonction `node-reconstruction`, présentée 5.4.2. L'identifiant du nœud à construire est contenu dans la ligne de *mappings* (*line*) passée en paramètre, on le note *line.*(*rnode.child.value*).

Enfin, si le nœud est de type expression, on construit sa valeur avec la fonction `instanciate--expression`. Cette fonction permet de construire le nœud XML correspondant au résultat de l'évaluation de l'expression. Pour cela, elle peut utiliser la fonction `node-reconstruction` pour construire des nœuds XML à partir des données stockées dans les tables du schéma générique. Lorsque l'expression contient une fonction d'agrégation, `instanciate-expression` utilise la partition passée en paramètre pour son évaluation.

5.5 Conclusion

La matérialisation des vues VIMIX est présentée dans ce chapitre. Elle permet de définir un **entrepôt de données** XML stocké dans un SGBD relationnel. L'entrepôt est défini comme une ensemble de vues VIMIX. L'ensemble de leurs spécifications constitue le schéma médiateur de l'entrepôt.

Les données extraites des sources sont stockées dans un **schéma générique**, en utilisant une méthode de méta modélisation. Les *mappings* qui spécifie les données extraites par les motifs sur les sources, les fragments et les jointures sont des méta données qui sont stockées dans des tables. Ces tables sont organisées en un graphe de *mappings*.

Notre architecture de stockage permet de maintenir **incrémentalement** les vues matérialisées, dans un contexte de sources non monitorées. Pour cela nous avons proposé des algorithmes permettant de maintenir l'entrepôt lorsque les données des sources sont **rafraîchies** et que la spécification des vues `évolue`.

Ces algorithmes permettent de maintenir incrémentalement l'entrepôt grâce à l'utilisation des identifiants pour les *mappings*. Le mécanisme utilisé permet de propager les modifications dans le graphe de *mappings* pour éviter de recalculer tous les *mappings* stockés.

Enfin, nous avons présenté les algorithmes qui permettent de construire le résultat des vues. En effet, les données stockées dans le SGBD relationnel nécessitent d'être recomposées pour fournir des données XML. Pour construire le résultat d'une vue. Les algorithmes proposés utilisent sa spécification, la table contenant les *mappings* de sa source et les tables du schéma générique.

Dans le chapitre suivant, nous présentons le système DaWaX que nous avons implémenté. Ce système permet de spécifier et construire un entrepôt de données XML défini par des vues VIMIX. Il propose une interface graphique qui facilite la spécification des vues.

Chapitre 6

Le système DaWaX

6.1 Introduction

Dans ce chapitre, nous présentons le système que nous avons développé pour implémenter notre modèle de vues vimix. Le système DaWaX (Da*ta*Wa*rehouse for* Xml) est un outil qui permet de spécifier, construire et maintenir un entrepôt de données xml.

Nous avons vu dans le chapitre 5 que la matérialisation des vues vimix permettait de construire un entrepôt de données xml. Le système DaWaX fournit une interface graphique qui permet de spécifier des vues vimix pour définir le schéma médiateur d'un entrepôt de données.

De plus, le système DaWaX permet de construire l'entrepôt de données à partir de sa spécification en utilisant un sgbd relationnel. Nous avons implémenter les algorithmes qui permettent d'extraire les données des sources pour alimenter et maintenir l'entrepôt de données.

Ce chapitre est organisé comme suit.

- Dans la section 6.2, nous présentons l'architecture du système DaWaX. Nous justifions aussi les choix des langages et outils utilisés pour l'implémentation du système.
- Dans la section 6.3, nous présentons l'interface graphique qui permet de spécifier les vues vimix. Cette interface permet au concepteur de l'entrepôt de données de spécifier des vues sans connaître la syntaxe de vimix. De plus, elle implémente les mécanismes présentés précédemment.
- Dans la section 6.4, nous présentons les algorithmes qui permettent d'extraire les données des sources pour alimenter l'entrepôt. Pour cela, il faut évaluer les axes de recherches spécifiés par les motifs sur les sources. Enfin, nous présentons l'interface graphique qui permet de maintenir les données de l'entrepôt.

6.2 Présentation générale

6.2.1 Architecture fonctionnelle

La Figure 6.1 présente l'architecture fonctionnelle de notre système. DaWaX est composé de trois interfaces qui permettent de définir, d'alimenter et d'interroger des sources de données XML multiples et hétérogènes.

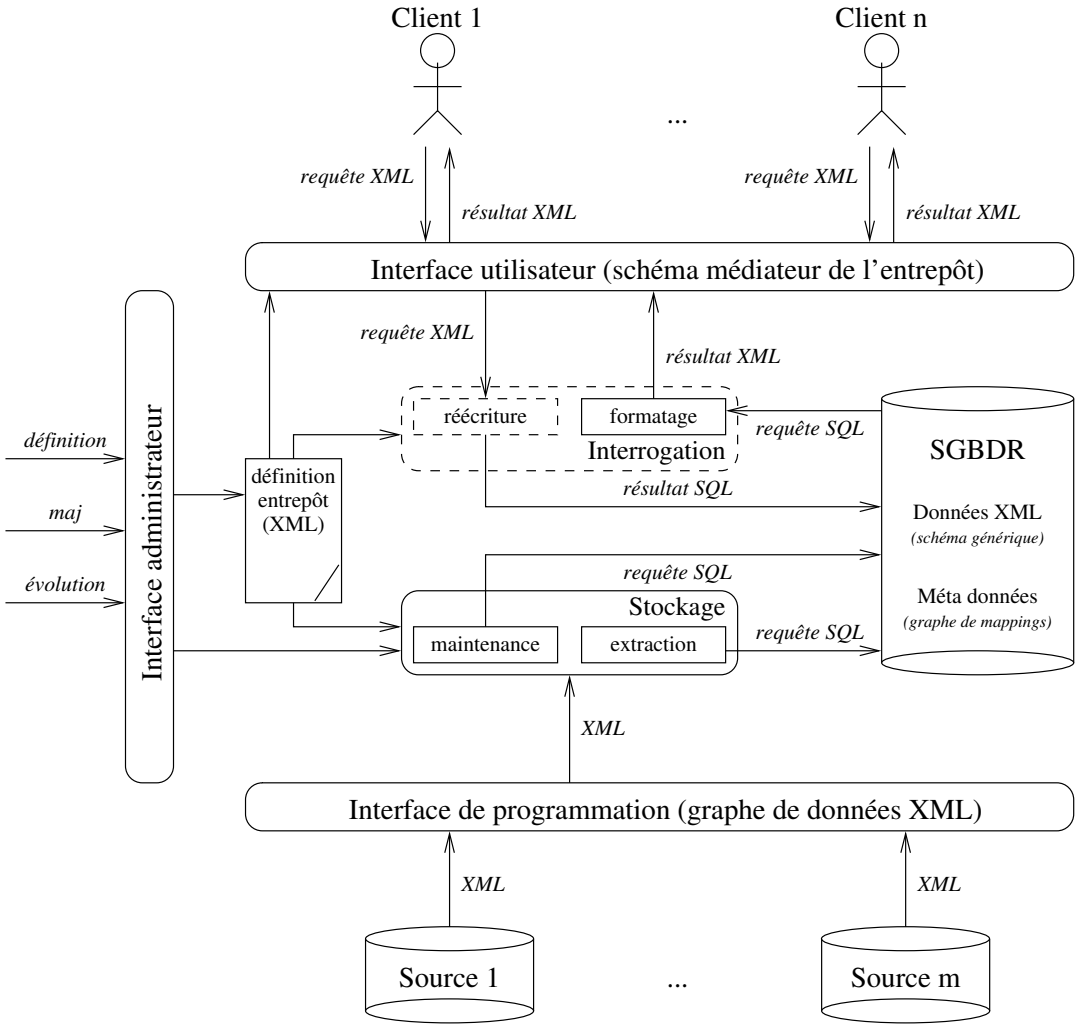

FIG. 6.1 – Architecture de DaWaX.

L'interface de programmation (*API*) fournit un graphe de données XML qui permet de représenter les données sources. La définition de ce graphe est présentée dans le chapitre 3 (§ 3.2, page 38).

L'interface administrateur permet de définir le schéma de l'entrepôt. Elle produit une spécification XML qui contient la définition des vues de l'entrepôt. Cette interface est également chargée d'alimenter et de maintenir l'entrepôt. Pour cela, elle utilise le composant `Stockage` qui permet d'extraire et de

maintenir les données de l'entrepôt. Les données sont stockées dans un SGBD relationnel, qui contient un schéma générique et un graphe de *mappings* qui sont présentés dans le chapitre 5 (§ 5.2, page 81). La maintenance de l'entrepôt (rafraîchissement et évolution) s'effectue avec des requêtes SQL de manipulation et de définition des données. Les algorithmes qui permettent de générer ces requêtes sont présentés dans le chapitre 5 (§ 5.3, page 98).

L'interface utilisateur permet d'interroger les données XML de l'entrepôt en utilisant le schéma médiateur. Les requêtes XML posées au système sont réécrites en requêtes SQL pour interroger le SGBD relationnel. Le résultat des requêtes SQL est formaté en XML, donnant ainsi l'illusion d'un système "tout XML". Les algorithmes permettant de construire le résultat d'une vue sont présentés dans le chapitre 5 (§ 5.4, page 107). Le traitement des requêtes utilise un mécanisme de réécriture qui permet de traduire une requête XML en SQL pour interroger le SGBD. Cette fonctionnalité qui apparaît en pointillé sur la figure, n'a pas été implémentée dans notre système. Cependant nous avons proposé des algorithmes qui permettent de réécrire les requêtes (§ E, page 165).

6.2.2 Implémentation

Nous avons implémenté le système DAWAX en utilisant le langage *Java* [Jav], dans sa version `1.4.2`. Le choix du langage *Java* présente les avantages suivants.

- C'est un langage bien connu et largement répandu. Il existe de nombreuses librairies qui facilite le développement d'applications, notamment des *parsers* pour XML.
- Les applications *Java* s'exécutent en utilisant une machine virtuelle, ce qui les rend indépendantes du système d'exploitation. Des machines virtuelles *Java* ont été développées pour la plupart des systèmes actuels, ce qui facilite la portabilité des applications *Java*.
- C'est un langage orienté objet.
- Les compilateurs *Java* sont gratuits.
- Enfin, *Java* permet de définir facilement des interfaces graphiques agréables à utiliser.

Les données de l'entrepôt sont stockées dans un SGBD relationnel. Nous avons choisi d'utiliser le système POSTGRESQL, dans sa version `7.3.2`. C'est SGBD relationnel/objet qui supporte la spécification du langage SQL. Nous avons choisi d'utiliser ce système car il est gratuit, performant et bien documenté. De plus, il fournit des *drivers jdbc* qui permettent de faire interagir le SGBD avec le langage *Java*.

L'interface graphique, que nous allons présenter dans ce chapitre, a été développée en utilisant les *packages* de la librairie *Swing*. Cette librairie présente une abstraction de l'interface graphique d'un système et permet de définir facilement des interfaces agréables à utiliser.

Enfin, nous avons utilisé les *packages Java* pour implémenter notre système de façon modulaire. Les principaux *packages* que nous avons développés sont les suivants.

- Le *package* `dawax` contient le point d'entrée de l'application (méthode `main`). Il permet de gérer la fenêtre principale de l'application.

- Le *package* `dawax.definition` contient les classes et méthodes qui permettent de définir l'interface graphique pour spécifier des vues VIMIX.
- Le *package* `dawax.management` contient les classes et méthodes qui permettent de construire, alimenter et maintenir l'entrepôt de données. Il utilise l'interface *jdbc* pour envoyer des requêtes au SGBD relationnel.
- Le *package* `dawax.xml.datamodel` contient l'implémentation du modèle de données que nous avons proposé (§ 3.2, page 38).
- Le *package* `dawax.xml.parser` contient le *parser* qui permet de construire un graphe de données XML. Il utilise le *parser* SAX (*Simple API for* XML) pour construire un graphe de données avec les classes de `dawax.xml.datamodel`.

6.3 Définition de vues VIMIX

La Figure 6.2 présente la partie de l'interface graphique de DAWAX qui permet de spécifier des vues VIMIX pour spécifier un entrepôt de données XML (onglet `XML`).

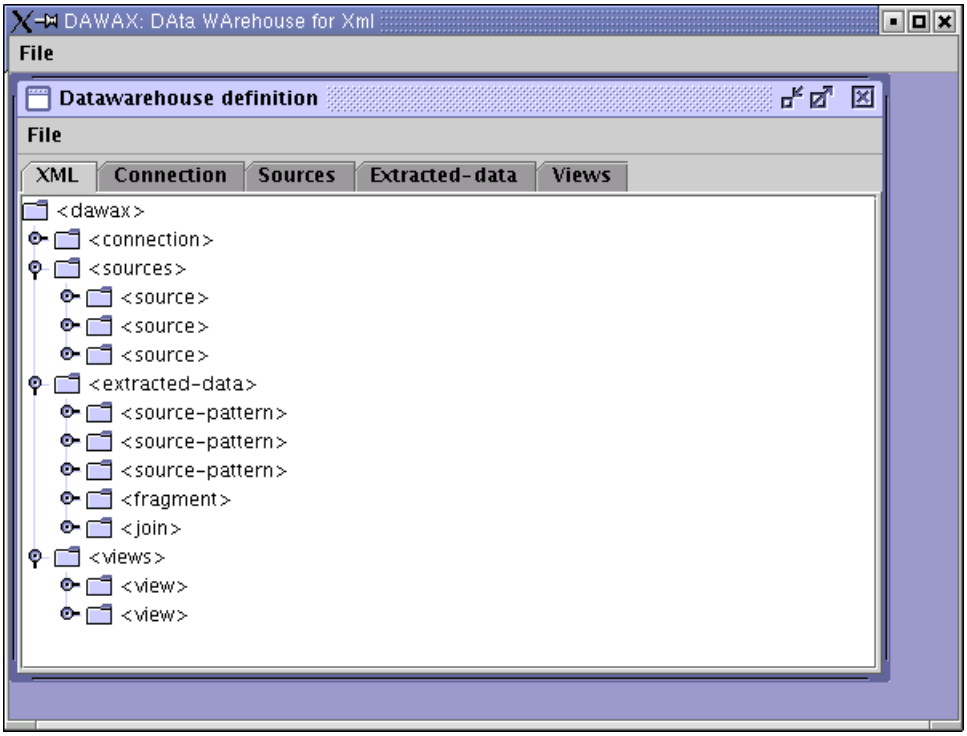

FIG. 6.2 – Ecran principal pour la spécification de vues VIMIX.

La fenêtre `Datawarehouse definition` est composée de plusieurs onglets. L'onglet `XML`, qui est visible sur la figure, affiche le document XML qui présente la spécification du système d'intégration. C'est ce document qui permet de définir l'entrepôt de données. Il est affiché sous la forme d'un arbre, à l'aide d'un composant *JTree* de la bibliothèque *Swing*.

Les autres onglets permettent de spécifier le système d'intégration. Ils sont présentés dans le reste de cette section. L'onglet `Connection` permet de spécifier les paramètres *jdbc* pour se connecter au SGBD. L'onglet `Sources` permet de spécifier les sources de données du système. L'onglet `Extracted-data` permet de spécifier les données à extraire dans les sources. Enfin, l'onglet `Views` permet de spécifier les vues qui définissent le schéma médiateur de l'entrepôt de données.

6.3.1 Connexion au SGBD relationnel

La Figure 6.3 présente la partie de l'interface graphique de DaWaX qui permet de spécifier les paramètres de connection utilisés par l'application pour se connecter au SGBD (onglet `Connection`).

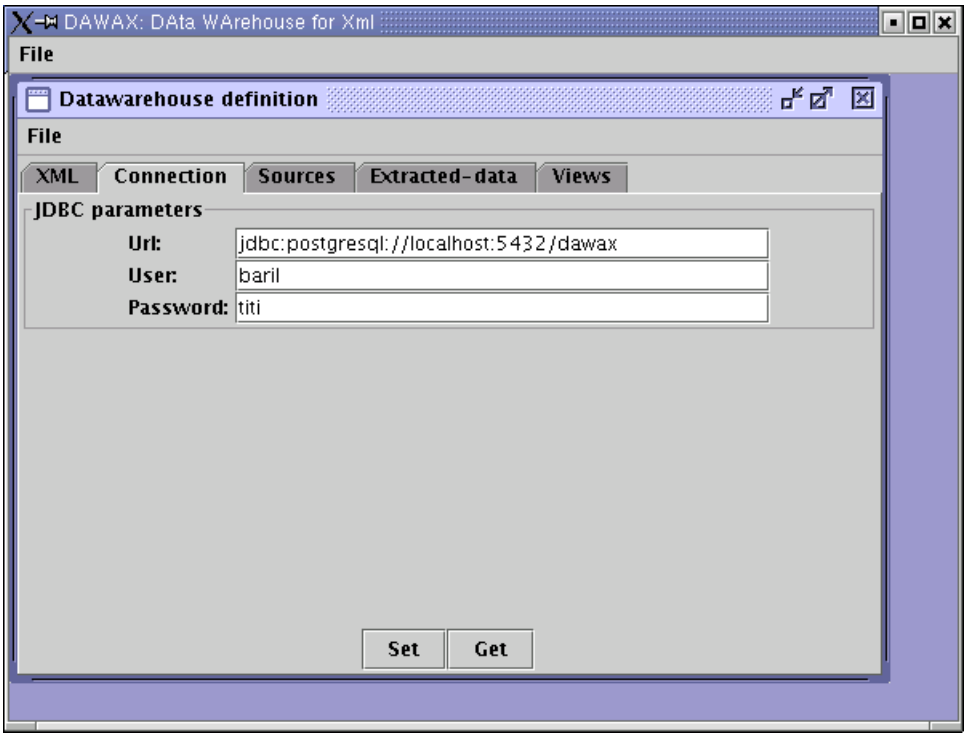

FIG. 6.3 – Ecran pour la spécification des paramètres de connexion au SGBD.

Le panneau `JDBC parameters` permet de spécifier les paramètres utilisés par l'interface de programation *jdbc*.

- Le champ `Url` indique l'adresse du serveur de base de données et le nom de la base.
- Le champ `User` indique l'utilisateur pour la connexion.
- Le champ `Password` indique le mot de passe correspondant à cet utilisateur.

6.3.2 Spécification des sources

La Figure 6.4 présente la partie de l'interface graphique de DaWaX permettant de spécifier les sources de données à intégrer dans notre système (onglet `Sources`).

L'onglet `Sources` est composé de plusieurs panneaux.

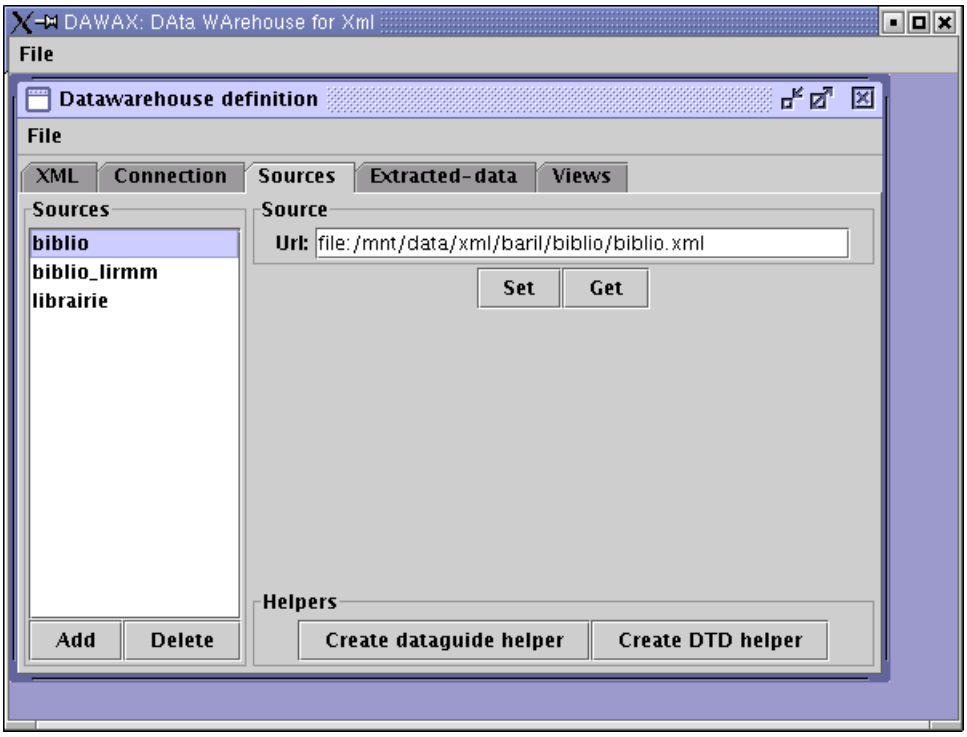

FIG. 6.4 – Ecran pour la spécification des sources de données.

- Le panneau `Sources` contient la liste des sources du système. Les boutons `add` et `delete` permettent respectivement d'ajouter et de supprimer une source à la spécification.
- Le panneau `Source` contient l'url de la source sélectionnée dans la liste. Dans la figure, la source sélectionnée est `biblio` et son `url` est `file :/mnt/data/xml/baril/biblio/biblio.xml`. Les boutons `set` et `get` permettent respectivement de mettre à jour l'url et d'afficher l'adresse de la source.
- Le panneau `Helpers` permet de gérer la création des assistants qui seront associés à une source de données. Les boutons `Create dataguide helper` et `Create dtd helper` permettent respectivement de créer une aide basée sur un *dataguide* ou une DTD pour la source sélectionnée. Ces assistants pourront être utilisés par l'interface de spécification des motifs utilisant cette source.

6.3.3 Spécification des données à extraire

La Figure 6.5 présente la partie de l'interface permettant de spécifier les données des sources à extraire (onglet `Extracted-data`). Nous avons vu que pour cela, on pouvait utiliser des motifs sur les sources, des fragments et des jointures. Le panneau `Extracted data` contient la liste des motifs, fragment et jointure spécifiés dans le système. Les boutons `Add pattern`, `Add fragment` et `Add join` permettent respectivement d'ajouter un motif, un fragment et une jointure. La partie droite de la fenêtre permet de spécifier le motif, le fragment ou la jointure sélectionnée dans la liste. Pour cela, on utilise les onglets `Source pattern`, `Fragment` et `Join` (en bas à droite).

Motifs sur les sources

La Figure 6.5 présente la partie de l'interface graphique de DaWaX permettant de spécifier des motifs sur les sources (onglet `Source pattern`).

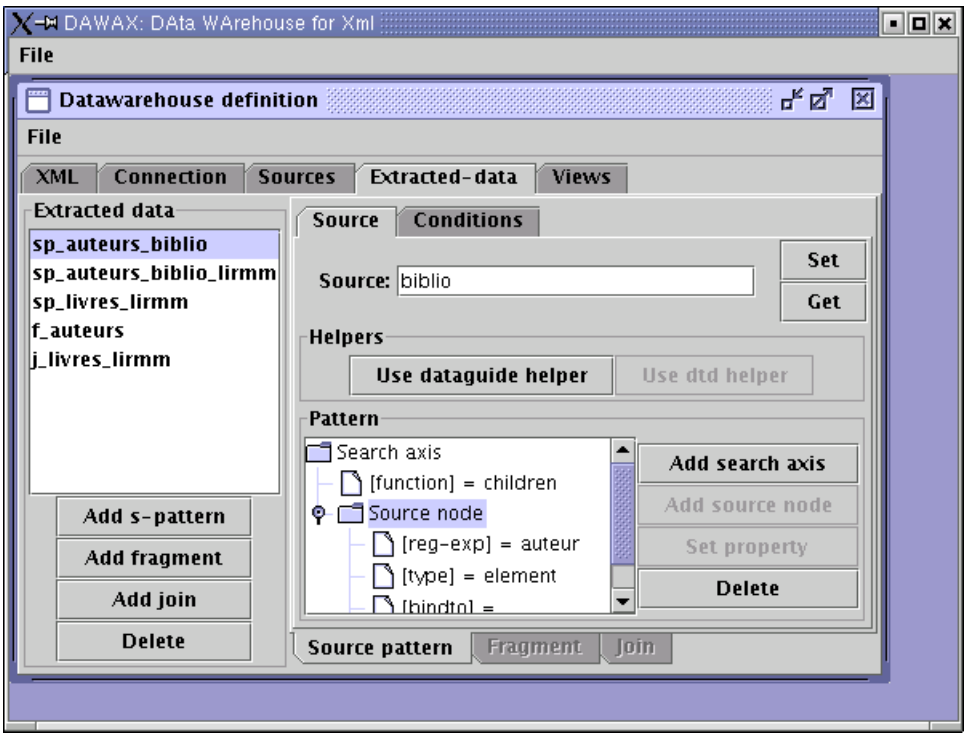

FIG. 6.5 – Ecran pour la spécification de motifs sur les sources.

L'onglet `Source pattern` est composé de plusieurs onglets.

- L'onglet `Source` permet de spécifier la source à utiliser. Le panneau `Pattern` permet de spécifier la forme à retrouver dans la source. Pour cela, il affiche un arbre qui affiche les éléments `search-axis` et `source-node`. Le panneau `Helpers` permet d'afficher un assistant pour la création du motif. Ici le bouton `Use dataguide helper` est actif car un assistant basé sur un *dataguide* a été créé pour cette source. Le bouton `Use dtd helper` est grisé car il n'y a pas d'assistant basé sur la DTD qui a été créé pour cette source.
- L'onglet `Conditions` permet de spécifier la conjonction de conditions associée au motif.

Fragments

La Figure 6.6 présente la partie de l'interface graphique de DaWaX permettant de spécifier des fragments (onglet `Fragment`).

L'onglet `Fragment` est composé de plusieurs fragments.

- L'onglet `Sources` permet de spécifier les sources de données extraites dont le fragment doit faire l'union. Pour cela, il affiche la liste des sources de données du fragment. Les boutons `Add` et `Delete` permettent d'ajouter et de supprimer une source.

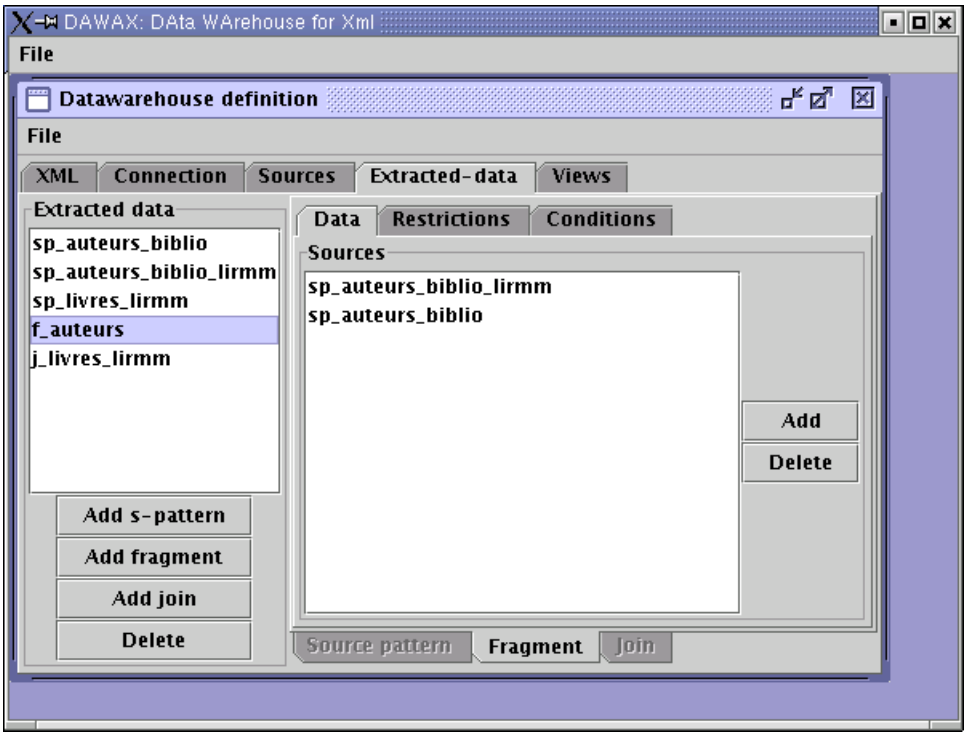

FIG. 6.6 – Ecran pour la spécification de fragments.

– L'onglet `Restrictions` permet de spécifier les restrictions sur les données extraites par le fragment.
– L'onglet `Conditions` permet de spécifier la conjonction de conditions associée au fragment.

Jointures

La Figure 6.7 présente la partie de l'interface graphique de DAWAX permettant de spécifier des jointures (onglet `join`).

L'onglet `Properties` permet de spécifier les propriétés de la jointure.
– Le champ `Left data` indique la source de données utilisée par la partie gauche de la jointure.
– Le champ `Left variable` indique la variable utilisée par la partie gauche de la jointure.
– Le champ `Right data` indique la source de données utilisée par la partie droite de la jointure.
– Le champ `Right variable` indique la variable utilisée par la partie droite de la jointure.

6.3.4 Spécification des vues

La Figure 6.8 présente la partie de l'interface graphique de DAWAX permettant spécifier les vues VIMIX (onglet `Views`).

L'onglet `Views` est composé de plusieurs panneaux. Le panneau `Views` contient la liste des vues spécifiées par le système. Les boutons `Add` et `Delete` permettent respectivement d'ajouter et de sup-

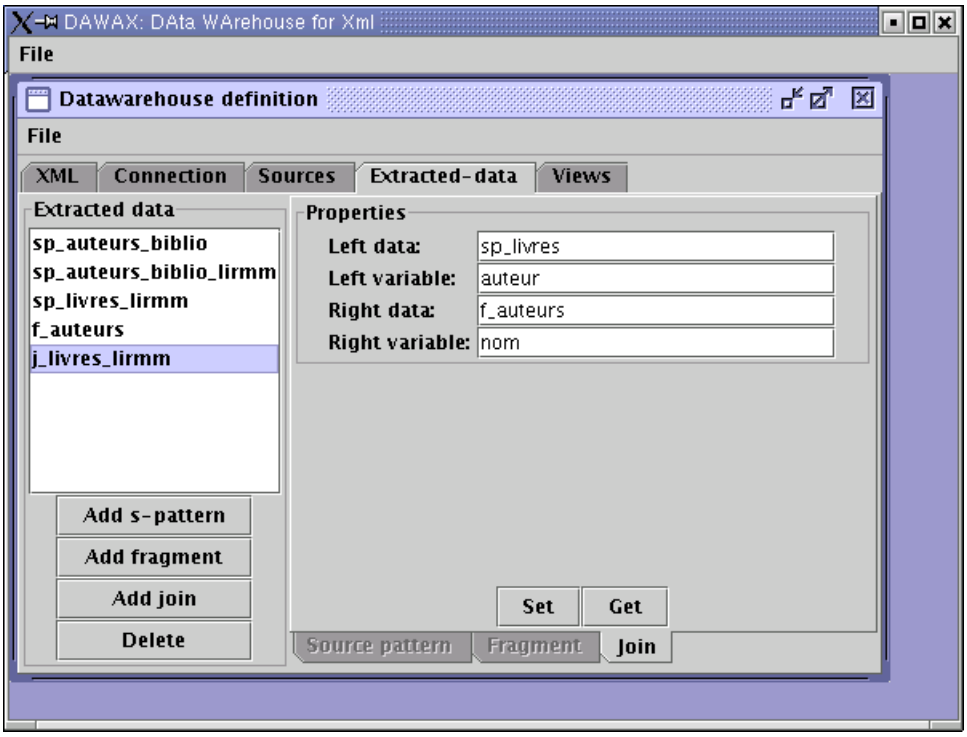

FIG. 6.7 – Ecran pour la spécification de jointures.

primer une vue au système d'intégration. Comme DAWAX permet de spécifier un entrepôt de données, les vues spécifiées seront matérialisées.

Le panneau `Properties` contient les propriétés de la vue sélectionnée dans la liste. Ici, la vue sélectionnée est `v_livres_lirmm`. Le champ `Source` indique la source de données utilisée par cette vue (ici `j_livres_lirmm`). Le champ `Order by` indique la variable à utiliser pour trier le résultat de la vue (ici `auteur`). Le champ `Group by` indique la variable utilisée pour créer un niveau de regroupement pour cette vue (ici `auteur`).

Le panneau `Result pattern` permet de spécifier la forme du résultat de la vue. Pour cela, il affiche l'arbre du résultat composé de nœuds `result-node`. Nous avions vu qu'il existait trois types de nœuds pour spécifier le résultat d'une vue.

- Les nœuds de type *element* sont affichés avec la notation suivante : `<value\>`.
- Les nœuds de type *attribute* sont affichés avec la notation suivante : `[value]`.
- Les nœuds de type *expression* sont les feuilles de l'arbre.

Les boutons sur la droite de ce panneau permettent de construire l'arbre spécifiant le résultat. Les boutons `Add element`, `Add attribute` et `Add expression` permettent d'ajouter un nouveau nœud dans la spécification du résultat de la vue, respectivement du type *element*, *attribute* et *expression*. Le bouton `Set value` permet de spécifier la valeur du nœud sélectionné dans l'arbre. Le bouton `Delete` permet de supprimer le nœud sélectionné.

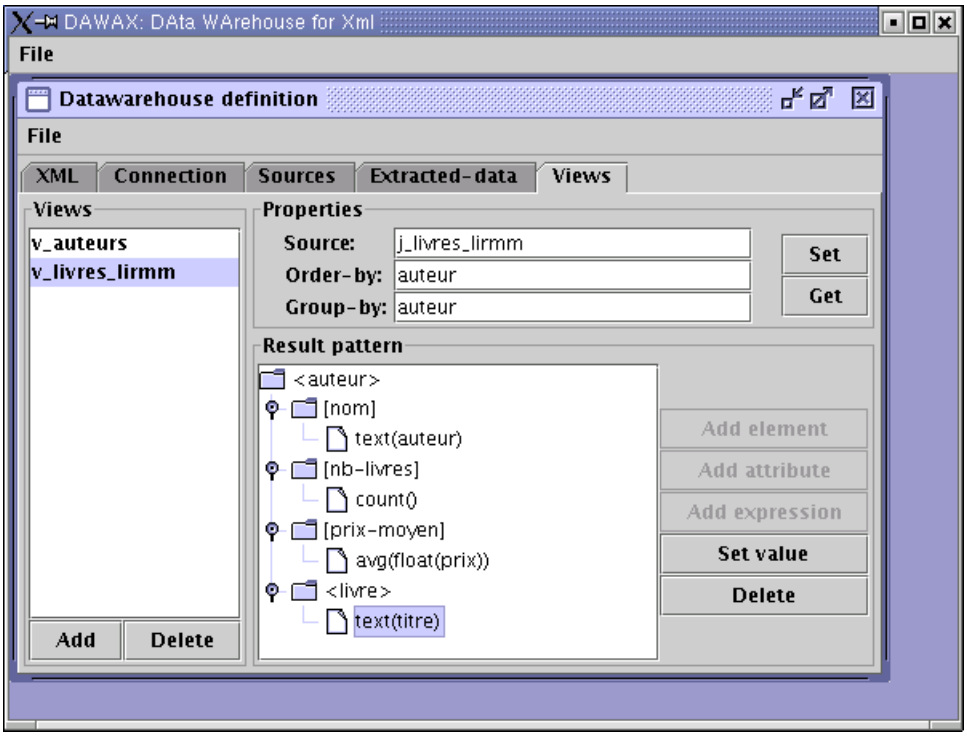

FIG. 6.8 – Ecran pour la spécification de vues.

Seuls les boutons autorisés pour le nœud sélectionné sont actifs. Ici le nœud sélectionné est l'expression `text(titre)`. Les boutons activés sont `Set value` et `Delete`. Comme un nœud de type *expression* doit être une feuille de l'arbre, les boutons permettant d'ajouter des nœuds ne sont pas activés.

6.4 Stockage des données

6.4.1 Extraction des données des sources

Nous présentons ici les techniques que nous avons mises en œuvre pour extraire les données des sources XML. Les algorithmes que nous avons implémentés dans notre système utilisent des opérations sur les listes que nous présentons ici. Pour extraire les données des sources, il est nécessaire d'évaluer les axes de recherches des motifs sur les sources. Pour cela, nous utilisons deux algorithmes `evaluate-axis` et `evaluate-node` que nous présentons plus bas.

Opérations sur les listes

La notation "`< >`" représente un constructeur de liste. Lorsqu'elle est utilisée sans argument, cette notation représente une liste vide. Lorsqu'elle est utilisée avec des arguments, elle construit une liste contenant les éléments passés en argument. Par exemple, "`<a,b>`" construit une liste contenant les éléments a et b. On peut utiliser récursivement le constructeur de liste, pour construire des listes de

listes. Par exemple, "`<<a,b>,<c>>`" permet de construire une liste composée de deux éléments, chacun représentant une liste. Le premier élément est une liste contenant les éléments a et b et le deuxième élément est une liste contenant l'élément c.

On peut concaténer des listes en utilisant l'opérateur "`+`". Cette opérateur permet de construire une nouvelle liste contenant tous les éléments des listes passées en argument. Pour illustrer cette opération, considérons les exemples suivants :

- `<> + <> = <>`,
- `<> + <a> = <a>`,
- `<a> + <b> = <a,b>`,
- `<a,b> + <c> = <a,b,c>`,

L'algorithme 10 permet d'effectuer une opération de multiplication sur des listes de listes. Le principe de cette opération est le suivant. Considérons deux listes de listes A et B. Chaque élément (liste) de A doit être concaténé avec chaque élément (liste) de B. Le résultat est une liste contenant toutes les concaténations des listes de A et B. Pour illustrer cette opération, considérons les exemples suivants :

- `multiplication(<>,<>) = <>`,
- `multiplication(<>,<<a>>) = <<a>>`,
- `multiplication(<<a>>,<<b>>) = <<a,b>>`,
- `multiplication(<<a,b>>,<<c>>) = <<a,c>,<b,c>>`,
- `multiplication(<<a,b>>,<<c,d>>) = <<a,c>,<a,d>,<b,c>,<b,d>>`,

Algorithme 10: `multiplication(`A`, `B`)`

Résultat : Renvoie la liste contenant le résultat de $A \times B$
si $A =<>$ **alors**
  retourner B ;
si $B =<>$ **alors**
  retourner A ;
$RES \leftarrow <>$;
pour chaque $LA \in A$ **faire**
  pour chaque $LB \in B$ **faire**
    $RES \leftarrow RES + < LA + LB >$;
retourner RES ;

Evaluation d'un axe de recherche

L'algorithme 11 permet d'évaluer un **axe de recherche** sur un nœud de la source de données appelé **nœud contextuel**. Pour cela les paramètres passées à la fonction sont `evaluate-axis` :

- l'axe de recherche à évaluer, noté $axis$,

– le nœud contextuel de cette évaluation, noté *cnode*.

Le résultat de cette évaluation renvoie toutes les instanciations possibles des variables contenues dans les nœuds de l'axe de recherche. Les instanciations sont renvoyées sous la forme d'une liste de listes. Chaque sous-liste représente une instanciation possible et contient la liste des valeurs des variables pour cette instanciation. Si le nœud contextuel ne respecte pas la spécification de l'axe de recherche, alors la fonction d'évaluation renvoie NULL.

Algorithme 11: `evaluate-axis(`*axis*`,`*cnode*`)`

Résultat : L'évaluation d'un axe de recherche renvoie la liste des instanciations possibles ou NULL si le nœud contextuel ne respecte pas la spécification

$RES \leftarrow <>$;
$C \leftarrow$ search-candidates(*axis*,*cnode*) ;
pour chaque $n \in$ `getChildrenElements(`*axis*`,"source-node")` **faire**
  $NL \leftarrow <>$;
  $ok \leftarrow false$;
  pour chaque $cn \in C$ **faire**
    $TEMP \leftarrow$ `evaluate-node(`n`,`cn`)` ;
    if $TEMP \neq$ NULL **then**
      $ok \leftarrow true$;
      $NL \leftarrow NL + TEMP$;
  si $\neg ok$ **alors**
    retourner NULL ;
  $RES \leftarrow RES \times NL$;
retourner `RES`;

L'évaluation d'un axe de recherche fonctionne de la manière suivante. Tout d'abord, nous calculons la liste des nœuds de la source de données qui sont candidats (dans la variable C). Ensuite, pour chaque nœud de l'axe de recherche, nous évaluons chaque nœud candidat de la source.

La fonction `search-candidates(`*axis*`,`*cnode*`)` permet d'effectuer la recherche de nœuds candidats d'un axe de recherche (*axis*), pour un nœud contextuel (*cnode*). Cette fonction utilise les opérations de navigation définies sur les nœuds XML de notre modèle de données (§ 3.2.3, page 43). Le principe de fonctionnement est le suivant : on récupère la valeur de l'attribut `function` de l'axe de recherche puis on applique l'opération de navigation correspondante pour calculer la liste des nœuds candidats.

Evaluation d'un nœud

L'algorithme 12 permet d'évaluer un **nœud appartenant à la spécification** d'un axe de recherche sur un **nœud contextuel** appartenant à la source de données. Pour cela, les paramètres passés à la fonction `evaluate-node` sont :

– la spécification du nœud à évaluer, notée *snode*,

– le nœud de la source à évaluer, noté *cnode*.

Le résultat de cette évaluation est la liste des instanciations possibles ou la valeur NULL si le nœud contextuel ne convient pas à la spécification.

La liste des instanciations possibles est représentée par une liste de listes ou chaque sous-liste correspond à une instanciation. Cette sous-liste contient les valeurs des variables pour cette instanciation.

Certaines spécifications de nœud sont liées à des variables mais ça n'est pas obligatoire. Si l'évaluation d'un nœud contextuel satisfait un nœud de spécification qui n'est pas associé à une variable, alors l'évaluation peut renvoyer une liste vide. Cette liste est vide si le nœud de spécification ne possède pas d'axe de recherche, ou que son axe de recherche renvoie une liste vide. La distinction entre une liste vide et la valeur NULL permet de différencier un nœud qui satisfait une spécification mais qui n'a pas de variables associées, d'un nœud qui ne satisfait pas une spécification.

Algorithme 12: `evaluate-node(`*snode*`,` *cnode*`)`

Résultat : L'évaluation d'un nœud renvoie la liste des instanciations possibles des variables du nœud ou NULL si le nœud ne repecte pas la spécification

```
// initialisation du résultat ;
RES ←<> ;
// vérifier le type du nœud s'il est précisé ;
si hasAttribute(snode,"type") alors
    si getAttribute(snode,"type") ≠ type(cnode) alors
        retourner NULL ;
// vérifier le nom du nœud ;
si getAttribute(snode,"name") ≠ getName(cnode) alors
    retourner NULL ;
// vérifier l'axe de recherche du nœud s'il existe ;
si hasChildElement(snode,"search-axis") alors
    axis ← getChildElement(snode,"search-axis") ;
    RES ← evaluate-axis(axis,cnode) ;
    si RES = NULL alors
        retourner NULL ;
// ajouter une valeur si le nœud est lié à une variable ;
si hasAttribute(snode,"bindto") alors
    si RES =<> alors
        RES ←<< cnode >> ;
    sinon
        pour chaque L ∈ RES faire
            L ←< cnode > +L ;
retourner RES ;
```

L'algorithme d'évaluation de la spécification d'un nœud sur un nœud contextuel est composé des étapes suivantes.

1. Le type du nœud contextuel est vérifié (si la spécification du nœud précise un type).
2. Le nom du nœud contextuel est vérifié.
3. Si la spécification du nœud contient un axe de recherche, alors le nœud de la source devient le nœud contextuel de l'évaluation de cet axe.
4. Le nœud contextuel est ajouté à la liste des instanciations (si la spécification lie le nœud à une variable).

6.4.2 Interface graphique pour la gestion des données

La Figure 6.9 présente la partie de l'interface graphique de DAWAX qui permet de gérer les données stockées dans l'entrepôt.

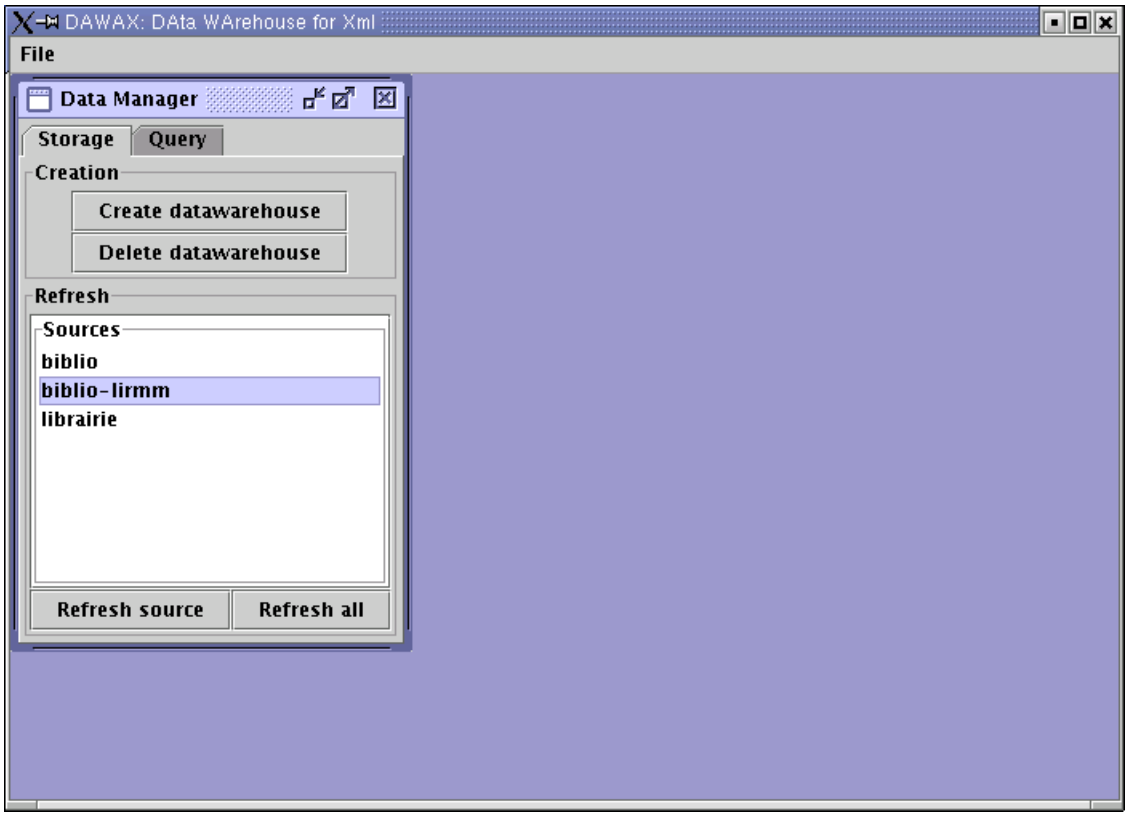

FIG. 6.9 – Ecran pour la gestion des données.

La fenêtre `Data Manager` est composé de deux panneaux. Le panneau `Construction` permet de lancer la construction de l'entrepôt de données (bouton `Create datawarehouse`) ou de le supprimer (bouton `Delete datawarehouse`).

Le panneau `Refresh` permet de rafraîchir les données stockées dans l'entrepôt. Il contient la liste des sources de données (panneau `Sources`) et deux boutons. Le bouton `Refresh source` permet de

rafraîchir la source de données sélectionnée dans la liste (ici `biblio-lirmm`). Le bouton `Refresh all` permet de rafraîchir toutes les sources de données de l'entrepôt.

6.5 Conclusion

Le système que nous avons implémenté est présenté dans ce chapitre. DaWaX (Da*ta*Wa*rehouse for* Xml) permet de définir un **entrepôt de données** xml. Pour cela, il implémente le mécanisme de vues que nous avons proposé : vimix. La matérialisation des vues spécifiées permettent de construire un entrepôt de données.

Le système DaWaX a été développé avec le langage *Java* et nous avons utilisé la bibliothèque *Swing* pour l'interface graphique. Le sgbd relationnel utilisé pour le stockage des données de l'entrepôt est le système PostGreSQL qui est performant et gratuit.

La construction de l'entrepôt se fait automatiquement à partir de sa spécification. Nous avons également proposé une interface graphique qui permet de maintenir les données de l'entrepôt en choisissant la source à rafraîchir.

Chapitre 7

Conclusion

7.1 Bilan et contributions

Dans ce mémoire, nous avons présenté une infrastructure de système d'intégration de données XML qui repose sur un modèle de vues : VIMIX (VIew Model for Integration of XML sources). Notre modèle de vues permet d'intégrer des données XML provenant de sources multiples et hétérogènes. L'utilisation d'XML comme langage commun nous a semblée pertinente car aujourd'hui de nombreux logiciels offrent la possibilité d'exporter leurs données en XML. La standardisation du langage par le W3C est un élément essentiel de son succès. L'objectif d'XML était de fournir un langage pour la représentation et l'échange de données sur le *Web*, on peut dire qu'il a été atteint. En effet, XML est largement utilisé dans les applications internet et intranet ce qui en fait un format idéal pour un langage commun dans un système d'intégration [BB01c].

Le schéma médiateur du système d'intégration offre une vue unifiée des sources de données hétérogènes. Il joue le rôle d'interface uniforme pour permettre aux utilisateurs d'utiliser les données intégrées sans se préoccuper de leur localisation ni de leur structure dans les sources. Nous avons défini le schéma médiateur de notre système comme un ensemble de vues VIMIX. Cette approche est généralement appelée *GAV* (*Global As View*) : la définition des vues sur les sources qui permet de construire une vue unifiée des données intégrées. Le modèle de vues que nous proposons, VIMIX, permet de restructurer les données des sources.

Le mécanisme de vues que nous avons proposé [BB00, BB03a] repose sur un modèle de données pour XML et un langage de requêtes. Notre modèle de données utilise un graphe pour représenter les données d'un document XML. Ce graphe est ordonné et comporte trois types de nœuds représentant les éléments, les attributs et les données textuelles d'un document XML. Il permet de représenter les liens de composition qui modélisent l'imbrication des éléments et des attributs. De plus, les liens de référence permettent de modéliser le mécanisme de partage des éléments offert par les attributs de type `IDREF` du langage XML.

Notre **langage de définition de vues** est basé sur le concept de *pattern-matching*. Les données à extraire sont définies à l'aide de variables dans des motifs sur les sources. D'un point de vue logique, ces

données peuvent être représentées dans des tables relationnelles. Chaque colonne de la table représente une variable et chaque ligne une instanciation de ces variables dans la source de données. Les fragments et les jointures permettent d'intégrer les données des sources avec des opérations d'union et de jointure. Enfin, la définition du résultat de la vue permet de restructurer les données intégrées. Pour cela, on peut créer de nouveaux éléments (ou attribut) et utiliser des niveaux de regroupement avec éventuellement des fonctions d'agrégation.

Nous avons proposé une méthode de **matérialisation** des vues VIMIX qui sépare les données extraites des sources et les méta données [BB03a]. Ces méta données (*mappings*) permettent d'associer les données des sources aux résultats des vues. Pour le stockage, nous utilisons un SGBD relationnel, ce qui permet de profiter des bonnes performances de ces systèmes. Les données XML des sources sont stockées grâce à la méta modélisation : nous avons défini un schéma générique qui permet de stocker des données XML dans un SGBD relationnel. Les méta données sont stockées dans un graphe de *mappings* : les nœuds de ce graphe sont les tables qui représentent les motifs, les fragments et les jointures utilisés pour extraire et intégrer les données des sources. La matérialisation des vues VIMIX permet de définir un **entrepôt de données**. Le schéma médiateur de l'entrepôt est spécifié par des vues VIMIX.

Notre méthode de stockage permet de **maintenir incrémentalement** l'entrepôt de données. Nous avons proposé des algorithmes qui prennent en compte le rafraîchissement des données et l'évolution de la spécification des vues qui définissent l'entrepôt. Ces algorithmes construisent des requêtes SQL pour mettre à jour le graphe de *mappings*. Nous avons considéré que les sources n'étaient pas monitorées, ce qui nous a semblé réaliste dans le contexte de sources XML sur le *Web*. La granularité des rafraîchissements est donc la modification d'une source de données. Les modifications sont propagées incrémentalement dans le graphe de *mappings*, c'est à dire que seules les données provenant de la source à rafraîchir sont mises à jour.

Enfin, pour valider notre approche, nous avons **développé le système** DAWAX (DA*ta*WA*rehouse for* XML) qui permet de définir, stocker et maintenir un entrepôt de données XML. Ce système utilise les méthodes et les algorithmes que nous avons présentés dans ce mémoire. Il est écrit en langage *Java* et utilise le SGBD relationnel POSTGRESQL. Les vues de l'entrepôt sont définies à travers une interface graphique qui propose un mécanisme d'aide pour écrire les motifs sur les sources [BB01a]. Le système DAWAX propose également une interface graphique pour la maintenance des données de l'entrepôt.

7.2 Perspectives

Nous avons identifié plusieurs perspectives pour améliorer et compléter le travail présenté dans ce mémoire. Tout d'abord, il serait intéressant de comparer expérimentalement les performances de notre

système avec les méthodes de stockage existantes. Ces mesures permettraient de quantifier l'apport en terme de performances de l'architecture que nous avons proposée pour le stockage et l'interrogation de notre entrepôt de données.

D'autres perspectives permettraient d'approfondir et d'enrichir les travaux que nous avons présentés. La sélection de vues à matérialiser permettrait d'améliorer le temps de réponse des requêtes. Dans le contexte des systèmes d'intégration à grande échelle, l'intégration sémantique permettrait d'intégrer (semi-)automatiquement un grand nombre de sources de données.

Sélection de vues à matérialiser

Le problème généralement appelé "sélection de vues à matérialiser" permet de définir la configuration d'un entrepôt de données étant donné un jeu de requêtes fréquemment posées à l'entrepôt. On entend ici par configuration un ensemble de vues matérialisées, qui permettent de minimiser le temps de réponse aux requêtes et le temps de maintenance de l'entrepôt. Ce problème a été largement étudié dans le contexte relationnel [TS97, GM99, BB03b]. Récemment, le système *LegoDB* [BFRS02, BFH+02] a proposé une méthode pour construire un schéma de méta modélisation, permettant de stocker des documents XML en fonction de requêtes fréquemment posées sur ces documents.

Notre approche permet de mettre en œuvre une politique de sélection de vues à matérialiser. En effet, en fonction d'un ensemble de requêtes frequemment posées à l'entrepôt, deux stratégies de matérialisation peuvent être considérées.

1. Matérialiser le résultat des requêtes SQL fréquemment réécrites pour interroger l'entrepôt de données.
2. Matérialiser les graphes de données XML fréquemment reconstruits pour répondre aux requêtes. Ces graphes pourraient stockés en utilisant soit le stockage à plat, soit une méthode de méta modélisation.

L'utilisation d'un SGBD relationnel pour le stockage de notre entrepôt de données peut de faciliter la mise en œuvre d'une politique de sélection de vues à matérialiser. En effet, le graphe de *mappings* que nous utilisons est similaire au graphe de matérialisation (*MVMG* : *Multi View Materialization Graph*) généralement utilisé dans le contexte relationnel. Dans le contexte de notre entrepôt de données XML, la formalisation du problème pourrait utiliser les paramètres suivants :

- V_{xml} un ensemble de vues définissant le schéma médiateur d'un entrepôt de données XML,
- S_{xml} un ensemble de sources de données XML utilisées par les vues,
- Q_{xml} un ensemble de requêtes XML,
- $f_u(s_{xml})$ une fonction qui détermine la fréquence de rafraîchissement d'une source de données,
- $f_a(q_{xml})$ une fonction qui détermine la fréquence d'interrogation d'une requête,
- R un schéma relationnel permettant de stocker l'entrepôt XML.

Le problème revient alors à trouver l'ensemble de vues sur R et/ou un ensemble de graphes de données XML à matérialiser, qui minimise la somme du temps de réponse d'un jeu de requêtes XML et du temps de maintenance des données stockées dans l'entrepôt.

Pour résoudre ce problème, il faudrait proposer un modèle de coûts qui permettrait de mesurer le coût d'exécution des requêtes et le coût de la maintenance incrémentale des sources.

Ensuite, il faudrait proposer une heuristique qui permettrait de déterminer quels sont les nœuds du graphe de *mappings* à matérialiser. Un mécanisme de cache peut également être proposé pour matérialiser certains graphe de données XML. Pour cela, on peut tenir compte d'un facteur de réutilisation comme celui que nous avons défini dans [BB03b].

Passage à l'échelle

Actuellement, les sources de données XML disponibles sur le *Web* sont de plus en plus nombreuses. Un critère important pour un système d'intégration est sa capacité à passer à l'échelle, c'est à dire intégrer des sources de données à l'échelle du *Web*. Avec le langage de définition de vues que nous proposons, il faut définir un motif sur chaque source de données à intégrer dans l'entrepôt. Cette tâche peut s'avérer fastidieuse lorsque les sources de données sont nombreuses, voir impossible dans un système à très grande échelle intégrant des milliers de sources.

Nous pourrions donc étendre le langage de vues, en introduisant le concept de "méta motif". Ces méta motifs permettraient de construire (semi)-automatiquement des motifs sur les sources de données à intégrer. Ces motifs ainsi construits permettraient d'extraire les données des sources et les *mappings* pour construire les vues. Cela permettrait donc d'intégrer automatiquement de nombreuses sources de données et ainsi d'assurer le passage à l'échelle du système. Pour cela, deux pistes nous semblent intéressantes à explorer.

La première consiste à exploiter les techniques développées en *datamining* pour la recherche de motifs fréquents. Ces techniques permettent d'extraire des structures qui apparaissent fréquemment dans des grandes bases de données semistructurées. Ces motifs fréquents pourraient être utilisés pour construire des méta motifs car ils fournissent une vue résumée des sources de données.

Ensuite, les techniques développées dans le domaine du TALN (Traitement Automatique du Langage Naturel) permettraient de définir des méta motifs en proposant un mécanismes d'intégration sémantique. Par exemple, Le système Xylème, propose un module d'intégration semi-automatique qui permet de construire des *mappings* entre les données des documents à intégrer et le schéma médiateur de l'entrepôt [RSV01] grâce à l'utilisation de fonctions lexicales. Nous pensons que l'utilisation de vecteurs conceptuels [SLP02, LPS02] permettrait de construire des motifs sur les sources à partir de méta motifs. Un vecteur conceptuel représente la sémantique d'un terme en lui attribuant des poids pour les différents concepts recensés dans un thésaurus. Des opérations sur les vecteurs permettraient de calculer la sémantique des éléments dans le méta motif et dans les sources de données. Les fonctions lexicales qui ont été définies en utilisant les vecteurs conceptuels permettraient de décider si un élément d'une source correspond à la spécification d'un méta motif.

Bibliographie

[AAC+01] Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors. *Proceedings of 27th International Conference on Very Large Data Bases, VLDB'2001*, Roma, Italy, September 11-14 2001. Morgan Kaufmann.

[AB91] Serge Abiteboul and Anthony J. Bonner. Object and Views. In Clifford and King [CK91], pages 238–247.

[ABC+00] Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors. *Proceedings of 26th International Conference on Very Large Data Bases, VLDB'2000*, Cairo, Egypt, September 10-14 2000. Morgan Kaufmann.

[Abi97] Serge Abiteboul. Querying semi-structured data. In Afrati and Kolaitis [AK97], pages 1–18. Invited talk.

[Abi99] Serge Abiteboul. On Views and XML. In PODS'1999 [POD99], pages 1–9. invited talk.

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[ACD01] Michel E. Adiba, Christine Collet, and Bipin C. Desai, editors. *Proceedings of the International Database Engineering & Applications Symposium, IDEAS'2001*, Grenoble, France, July 16-18 2001. IEEE Computer Society.

[ACFR01] Serge Abiteboul, Sophie Cluet, Guy Ferran, and Marie-Christine Rousset. The Xyleme Project. Gemo Report 248, INRIA, November 2001.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AJEM02] Lina Al-Jadir and El-Moukaddem. F2/XML : Storing XML Documents in Object Database. In Bellahsène et al. [BPR02], pages 108–116.

[AK97] Foto N. Afrati and Phokion G. Kolaitis, editors. *Proceedings of the 5th International Conference on Database Theory*, Delphi, Greece, January 8-10 1997. Springer.

[AMM99] Paolo Atzeni, Alberto O. Mendelzon, and Giansalvatore Mecca, editors. *The World Wide Web and Databases, International Workshop WebDB'98*, volume 1590 of *Lecture Notes in Computer Science*, Valencia, Spain, 1999. Springer.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Librairies*, 1(2) :68–88, September 1997.

[Are01] Walid G. Aref, editor. *Proceedings of the ACM SIGMOD Conference*, Santa Barbara, CA, May 21-24 2001. Electronic Proceedings (available to SIGMOD members at *http ://www.acm.org/sigmod/sigmod01/eproceedings*).

[Bar99] Xavier Baril. GEDOO : Historisation dans les entrepots de données. Rapport de DEA, Université Paul Sabatier (Toulouse III), 1999.

[BB99] Catriel Beeri and Peter Buneman, editors. *Proceedings of the 7th International Conference on Database Theory*, Jerusalem, Israel, January 10-12 1999. Springer.

[BB00] Xavier Baril and Zohra Bellahsène. A View Model for XML Documents. In Patel et al. [PCPdC00], pages 429–441.

[BB01a] Xavier Baril and Zohra Bellahsène. A Browser for Specifying XML Views. In Wang et al. [WPJ01], pages 164–174.

[BB01b] Xavier Baril and Zohra Bellahsène. Conception et implémentation d'un entrepôt XML. Actes de la Journée de Travail Bi-Thématique du GDR-PRC I3, 13 Décembre 2001.

[BB01c] Zohra Bellahsène and Xavier Baril. XML et les systèmes d'intégration de données. *Ingénierie des systèmes d'information*, 6(3) :11–32, 2001.

[BB03a] Xavier Baril and Zohra Bellahsène. Designing and Managing an XML Warehouse. In Akmal Chaudhri, editor, *XML Data Management : Native XML and XML-Enabled Database Systems*, chapter 16. Addison-Wesley, 2003.

[BB03b] Xavier Baril and Zohra Bellahsène. Selection of Materialized Views : a Cost Based Approach. In Eder and Missikoff [EM03].

[BDA94] *Actes des 10e Journées Bases de Données Avancées*, Clermont-Ferrand, France, 1994.

[BDA00] *Actes des 16e Journées Bases de Données Avancées*, Blois, France, 2000.

[BEA] BEA. Web site. *http ://www.bea.com.*

[Bel00] Zohra Bellahsène. Updates and object-generating views in ODBS. *Data and Knowledege Engineering*, 34(2) :125–163, August 2000.

[BFH+02] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. LegoDB : Customizing Relational Storage for XML Documents. In VLDB'2002 [VLD02], pages 1091–1094. demo session.

[BFRS02] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. From XML Schema to Relations : A Cost-based Approach to XML Storage. In ICDE'2002 [ICD02].

[BG02] Edgard Ivan Benitez Guerrero. *Infrastructure adaptable pour l'évolution des entrepôt de données.* PhD thesis, Université Joseph Fourier (Grenoble I), 2002.

[BJ93] Peter Buneman and Sushil Jajodia, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 26-28 1993. ACM Press.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. In Zaniolo [Zan86], pages 61–71.

[BPR02] Zohra Bellahsène, Dilip Patel, and Colette Rolland, editors. *Proceedings of 8th International Conference on Object Oriented Information Systems, OOIS'2002*, Montpellier, France, September 2-5 2002. Springer.

[BR94] Zohra Bellahsène and Hugues Ripoche. An Object-Oriented Database for the Management of of Genetic Sequences. In BDA'94 [BDA94].

[Bru01] Emmanuel Bruno. Documents XML : un modèle et une algèbre. *Ingénierie des systèmes d'information*, 6 :73–93, 2001.

[BTW03] *Proceedings of the Datenbanksysteme für Business, Technologie und Web*, LNI, Liepzig, Deutschland, February 26-28 2003. GI.

[CAM02] Gregory Cobena, Serge Abiteboul, and Amélie Marian. Detecting Changes in XML Documents. In ICDE'2002 [ICD02].

[CCD+98] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL : A Graphical Language for Querying and Reshaping XML Documents. In QL'98 [QL'98]. Electronic edition at *http ://www.w3.org/TandS/QL/QL98/*.

[CCS00] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On Wrapping Query Languages and Efficient XML Integration. In Chen et al. [CNB00], pages 141–152. SIGMOD Record 29(2).

[CGMH+94] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS Project : Integration of Heterogeneous Information Sources. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, October 1994.

[CIK95] *Proceedings of the 1995 International Conference on Information and Knowledge Management (CIKM)*, Baltimore, Maryland, November 28 - December 2 1995. ACM.

[CK91] James Clifford and Roger King, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, May 29-31 1991. ACM Press.

[CM00] Stephano Cerri and Daniele Maraschi, editors. *Proceedings of WITREC'2000*, Montpellier, France, 2000.

[CNB00] Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors. *Proceedings of the ACM SIGMOD Conference*, Dallas, Texas, May 16-18 2000. ACM. SIGMOD Record 29(2).

[Col99] Christine Collet, editor. *Actes des 15^e Journées Bases de Données Avancées*, Bordeaux, France, 1999.

[CPC95] Catherine Comparot-Poussier and Claude Chrisment. Hyperbase pour la Gestion Electronique de Documents Techniques. *Ingénierie des systèmes d'information*, 2(5) :533–570, 1995.

[CRF00] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt : An XML Query Language for Heterogeneous Data Sources. In Suciu and Vossen [SV00], pages 1–25. Invited Paper.

[CS95] Michael J. Carey and Donovan A. Schneider, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 22-25 1995. ACM Press.

[CS02] Claude Chrisment and Florence Sèdes. Towards a unified view of media annotation. *Ingénierie des systèmes d'information*, 7(1) :45–63, 2002.

[CW91] Stephano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In Lohman et al. [LSC91], pages 577–589. Best paper session.

[DFF+98] Alan Deutsch, May F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. XML-QL. In QL'98 [QL'98]. Electronic edition at *http ://www.w3.org/TandS/QL/QL98/*.

[DFF+99] Alan Deutsch, May F. Fernandez, Daniela Florescu, Alon Y. Levy, David Maier, and Dan Suciu. Querying XML Data. *IEEE Data Engineering Bulletin*, 22(3) :27–34, 1999.

[DFG99] Alex Delis, Christos Faloutos, and Shahram Ghandeharizadeh, editors. *Proceedings of the ACM SIGMOD Conference*, Philadelphia, Pennsylvannia, June 1-3 1999. ACM Press.

[DHW01a] Denise Draper, Alon Halevy, and S Daniel Weld. The Nimble Data Integration System. In ICDE'2001 [ICD01], pages 155–160.

[DHW01b] Denise Draper, Alon Halevy, and S Daniel Weld. The Nimble Integration Engine. In Aref [Are01]. Industrial Track Papers.

[EM03] Johann Eder and Michele Missikoff, editors. *Proceedings of 15th International Conference on Advanced Information Systems, CAISE'2003*, Klagenfurt/Verden, Austria, June 16-20 2003. Springer LNCS.

[Eno] Enosys Software. Web site. *http ://www.enosysmarkets.com.*

[eXm] e-XMLMedia. Web site. *http ://www.e-xmlmedia.fr/index.htm.*

[FK99] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3) :27–34, 1999.

[FS98] May F. Fernandez and Dan Suciu. A Query Language for XML. In QL'98 [QL'98]. Electronic edition at *http ://www.w3.org/TandS/QL/QL98/*.

[Gar02] Georges Gardarin. *XML : des bases de données aux services Web.* Dunod, 2002.

[GIFF+99] Zachary G Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S Weld. An Adaptative Query Execution System for Data Execution. In Delis et al. [DFG99], pages 299–310.

[Gio92] Wiederhold Gio. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3) :38–49, March 1992.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views : Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2) :3–18, 1995.

[GM99] Himanshu Gupta and Inderpal Singh Mumick. Selection of Views to Materialize Under a Maintenance Cost Constraint. In Beeri and Buneman [BB99], pages 453–470.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In Buneman and Jajodia [BJ93], pages 157–166.

[GMT02] Georges Gardarin, Antoine Mensch, and Antoine Tomasic. An Introduction of the e-XML Data Integration Suite. In Jensen et al. [JJP+02].

[GMUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems, The Complete Book*. Prentice Hall, 2002.

[GMW99] Roy Goldman, Jason McHugh, and Jennifer Widom. From Semistructured Data to XML : Migrating the Lore Data Model and Query Language. In WEBDB'99 [WEB99], pages 25–30.

[GMW00] Roy Goldman, Jason McHugh, and Jennifer Widom. From Semistructured Data to XML : Migrating the Lore Data Model and Query Language. *Markup Language : Theorie and Practice*, 2(2), 2000.

[GW97] Roy Goldman and Jennifer Widom. DataGuides : Enabled Query Formulation and Optimization in Semistructured Databases. In Jarke et al. [JCD+97], pages 436–445.

[Hal00] Alon Y. Halevy. Theory of Answering Queries Using Views. *SIGMOD Record*, 29(1) :40–47, March 2000.

[Hal01] Alon Y. Halevy. Answering queries using views : A survey. *The VLDB Journal*, 10(4) :270–294, 2001.

[Hal03] Alon Halevy. Data Integration : A Status Report. In BTW'2003 [BTW03], pages 24–29.

[ICD00] *Proceedings of the 16th International Conference on Data Engineering, ICDE'2000*, San Diego, California, February 28 - March 3 2000. IEEE Computer Society.

[ICD01] *17th International Conference on Data Engineering, ICDE'2001*, Heidelberg, Germany, April 2-6 2001. IEEE Computer Society.

[ICD02] *18th International Conference on Data Engineering, ICDE'2002*, San José, California, February 26 - March 1 2002. IEEE Computer Society.

[IDD01] Manolescu Ioana, Florescu Daniela, and Kossmann Donald. Pushing XML Queries inside Relational Databases. Technical Report 4112, INRIA, January 2001.

[Jav] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.4.2, API Specification. http ://java.sun.com/j2se/1.4.2/docs/api/index.html.*

[JCD+97] Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors. *Proceedings of 23rd International Conference on Very Large Data Bases, VLDB'1997*, Athens, Greece, August 25-29 1997. Morgan Kaufmann.

[JJP+02] Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors. *Proceedings of the 8th International Conference on Extending Database Technology*, volume 2287 of *Lecture Notes in Computer Science*. Springer, 2002.

[KM99] C. Kanne and G. Moerkotte. Efficient Storage of XML data. Technical Report 899, Mannheim University, 1999.

[KM00] Carl-Christian Kanne and Guido Moerkotte. Efficient Storage of XML Data. In ICDE'2000 [ICD00], page 198.

[LM00] Jacques Le Maitre. Classification des langages pour XML. In Cerri and Maraschi [CM00], Invited Paper.

[LPS02] Mathieu Lafourcade, Violaine Prince, and Didier Schwab. Vecteurs conceptuels et structuration émergente de terminologies. *Traitement Automatiques des Langues (TAL)*, 43(1) :43–72, 2002.

[LQVC95] Pierre-Yves Lambolez, Jean-Pierre Queille, Jean-François Voidrot, and Claude Chrisment. EXREP : a generic rewriting tool for textual information extraction. *Ingénierie des systèmes d'information*, 3(4) :471–487, 1995.

[LS00] Hartmut Liefke and Dan Suciu. XMILL : An Efficient Compressor for XML Data. In Chen et al. [CNB00], pages 153–164. SIGMOD Record 29(2).

[LSC91] Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors. *Proceedings of 17th International Conference on Very Large Data Bases, VLDB'1991*, Barcelona, Spain, September 3-6 1991. Morgan Kaufmann.

[LVPV99] Bertram Ludäscher, Pavel Velhikov, Yannis Papakonstantinou, and Victor Vianu. View Definition and DTD Inference for XML. In *Proceedings of the Post-ICDT Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israël, 1999.

[MAA+00] Laurent Mignet, Serge Abiteboul, Sébastien Ailleret, Bernd Amann, Amélie Marian, and Mihai Preda. Acquiring XML pages for a Webhouse. In BDA'2000 [BDA00], pages 241–263.

[MACM01] Amélie Marian, Serge Abiteboul, Gregory Cobena, and Laurent Mignet. Change-Centric Management of Versions in an XML Warehouse. In Apers et al. [AAC+01], pages 581–590.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore : A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3) :54–66, September 1997.

[Mai98] David Maier. Database Desiderata for an XML Query Language. QL'98 - Query Languages 1998, electronic proceedings, December 3-4 1998. *http ://www.w3.org/TandS/QL/QL98/pp/maier.html*.

[May01] Wolfgang May. LoPiX : A System for XML Data Integration and Manipulation. In Apers et al. [AAC+01], pages 241–250.

[MBV03] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The XML Web : a First Study. In WWW'2003 [WWW03].

[MFK+00] Ioana Manolescu, Daniela Florescu, Donald Kossmann, Florian Xhumari, and Dan Olteanu. Agora : Living with XML and Relational. In Abbadi et al. [ABC+00], pages 623–626. demo session.

[MFK01] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML Queries on Heterogeneous Data Sources. In Apers et al. [AAC+01], pages 241–250.

[Mic99] Alain Michard. *XML - Langage et applications*. Eyrolles, 1999.

[MIX] MIX Mediator System. Web site of the project. *http ://www.npaci.edu/DICE/mix-system.html*.

[NACP01] Benjamin Nguyen, Serge Abiteboul, Gregory Cobéna, and Mihai Preda. Monitoring XML Data on the Web. In Aref [Are01]. Electronic Proceedings (available to SIGMOD members at *http ://www.acm.org/sigmod/sigmod01/eproceedings*).

[Nam99] Namespaces in XML, 14 Janury 1999. W3C Recommandation, *http ://www.w3.org/TR/1999/REC-xml-names-19990114*.

[NH99] Simon North and Paul Hermans. *XML*. Le programmeur. Campus Press, 1999.

[PBE95] Evaggelia Pitoura, Omran Bukhres, and Ahmed Elmagarmid. Object Orientation in Multidatabase Systems. *ACM Computing Surveys*, 27(2) :141–195, june 1995.

[PCPdC00] Dilip Patel, Islam Choudhury, Shushma Patel, and Sergio de Cesare, editors. *Proceedings of 6th International Conference on Object Oriented Information Systems, OOIS'2000*, London, UK, December 18-20 2000. Springer.

[PDI96] *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 18-20 1996. IEEE Computer Society.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jenifer Widom. Object Exchange Across Heterogenous Information Sources. In Yu and Chen [YC95], pages 251–260.

[POD99] *Proceedings of the Eighteenth Symposium on Principles Of Database Systems, PODS'1999*, Philadelphia, Pennsylvania, May 31 - June 2 1999. ACM Press.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing. In PDIS'1996 [PDI96], pages 158–159.

[QL'98] *The Query Language Workshop, QL'98*, Boston, Massachussets, December 3-4 1998. Electronic edition at *http ://www.w3.org/TandS/QL/QL98/*.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). In QL'98 [QL'98]. Electronic edition at *http ://www.w3.org/TandS/QL/QL98/*.

[RSV01] Chantal Reynaud, Jean-Pierre Sirot, and Dan Vodislav. Semantic Integration of Heterogeneous XML Data Sources. In Adiba et al. [ACD01], pages 199–208.

[SC99] Jérôme Siméon and Sophie Cluet. Using YAT to build a Web Server. In Atzeni et al. [AMM99].

[SGN99] Fei Sha, Georges Gardarin, and Laurent Némirovski. Managing Semistructured Data in Object-Relational DBMS. In Collet [Col99], pages 101–115.

[Sim00] Jérôme Siméon. Data Integration with XML : A Solution for Modern Web Applications. Lecture at Temple University, March 2000.

[SLP02] Didier Schwab, Mathieu Lafourcade, and Violaine Prince. Antonymy and Conceptual Vectors. In *Proceedings of COLING'2002*, pages 904–910, Taipei, Taiwan, August 2002.

[SV00] Dan Suciu and Gottfried Vossen, editors. *Proceedings of WebDB'2000 (Selected Papers)*, Dallas, Texas, May 18-19 2000. Springer.

[Tes00] Olivier Teste. *Modélisation et Manipulations des entrepôts de données complexes et historisées*. PhD thesis, Université Paul Sabatier (Toulouse III), 2000.

[TS97] Dimitri Theodoratos and Timos K. Sellis. Data Warehouse Configuration. In Jarke et al. [JCD+97], pages 126–135.

[UW97] Jeffrey D. Ullman and Jennifer Widom. *A first course in database systems*. Prentice-Hall, 1997.

[Vel02] Pierangelo Veltri. *Un système de vues pour les données XML du Web : conception et implantation*. PhD thesis, Université Paris XI, 2002.

[VLD02] *Proceedings of 28th International Conference on Very Large Data Bases, VLDB'2002*, Hong Kong, China, August 20-23 2002. Morgan Kaufmann.

[W3Ca] World Wide Web Consortium. Web site of the consortium. *http ://www.w3.org*.

[W3Cb] About the World Wide Web Consortium (W3C). Web site of the consortium. *http ://www.w3.org/Consortium*.

[WEB99] *Proceedings of WebDB'1999*, Philadelphia, Pennsylvania, June 1999.

[Wid95] Jennifer Widom. Research Problems in Data Warehousing. In CIKM'1995 [CIK95], pages 25–30. invited talk.

[WPJ01] Yingxu Wang, Shushma Patel, and Ronald Johnston, editors. *Proceedings of 7th International Conference on Object Oriented Information Systems, OOIS'2001*, Calgary, Canada, August 27-29 2001. Springer.

[WWW03] *Proceedings of WWW'2003*, Budapest, Hungary, May 20-24 2003. ACM.

[XLi] XLink.

[XML98] Extensible Markup Language (XML) 1.0, 10 February 1998. W3C Recommendation, *http ://www.w3.org/TR/1998/REC-xml-19980210*.

[XML00] Extensible Markup Language (XML) 1.0 (Second Edition), 6 October 2000. W3C Recommendation, *http ://www.w3.org/TR/2000/REC-xml-20001006*.

[XML01a] XML Schema Part 0 : Primer, 2 May 2001. W3C Recommendation, *http ://www.w3.org/TR/2001/REC-xmlschema-0-20010502*.

[XML01b] XML Schema Part 1 : Structures, 2 May 2001. W3C Recommendation, *http ://www.w3.org/TR/2001/REC-xmlschema-1-20010502*.

[XML01c] XML Schema Part 2 : Datatypes, 2 May 2001. W3C Recommendation, *http ://www.w3.org/TR/2001/REC-xmlschema-2-20010502*.

[XML03] Extensible Markup Language (XML) 1.1, 15 October 2003. W3C Candidate Recommendation, *http ://www.w3.org/TR/2000/CR-xml11-20021015*.

[XPa99] XML Path Language (XPath) 1.0, 16 November 1999. W3C Recommandation, *http ://www.w3.org/TR/1999/REC-xpath-19991116*.

[XPo] Xml pointer language (xpointer) 1.0.

[XQu03] XQuery 1.0 : An XML Query Language, 22 August 2003. W3C Working Draft, *http ://www.w3.org/TR/2002/WD-xquery-20030822*.

[XSL99] XSL Transformation (XSLT) 1.0, 16 November 1999. W3C Recommendation, *http ://www.w3.org/TR/1999/REC-xslt-19991116*.

[XSL01] Extensible Stylesheet Language (XSL) 1.0, 15 October 2001. W3C Recommendation, *http ://www.w3.org/TR/2001/REC-xsl-20011015*.

[Xyla] Xyleme. Web site of the start-up. *http ://www.xyleme.com*.

[Xylb] Xyleme : A Dynamic Warehouse for XML Data of the Web. Web site of the project. *http ://www-rocq.inria.fr/xyleme*.

[YASU01] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel : a Path-Based Approach to Storage and Retrieval of XML documents using relational databases. *ACM Transactions On Internet Technology, TOIT*, 1(1) :110–141, August 2001.

[YC95] Philip S. Yu and Arbee L. P. Chen, editors. *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, March 6-10 1995. IEEE Computer Society.

[Zan86] Carlo Zaniolo, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 28-30 1986. ACM Press.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In Carey and Schneider [CS95], pages 316–327.

Liste des tableaux

Liste des figures

Annexes

Annexe A

VIMIX : DTD validant la spécification d'un schéma médiateur

```
<!-- mediated schema -->
<!ELEMENT mediated-schema (sources, extracted-data, views) >
<!ELEMENT sources (source+) >
<!ELEMENT extracted-data (source-pattern+, (fragment | join)*) >
<!ELEMENT views (view+) >
<!-- source -->
<!ELEMENT source EMPTY >
<!ATTLIST source
          id  ID    #REQUIRED
          url CDATA #REQUIRED >
<!-- source pattern -->
<!ELEMENT source-pattern (search-axis, conditions?) >
<!ATTLIST source-pattern
          name   ID    #REQUIRED
          source IDREF #REQUIRED
>
<!-- search axis -->
<!ELEMENT search-axis (source-node+) >
<!ATTLIST search-axis
          function CDATA #REQUIRED
>
<!-- source node -->
<!ELEMENT source-node (search-axis?) >
<!ATTLIST source-node
          reg-expression CDATA #REQUIRED
          type           CDATA #IMPLIED
          bindto         CDATA #IMPLIED
>
<!-- fragment -->
<!ELEMENT fragment (conditions?, restrictions*) >
<!ATTLIST fragment
          name ID     #REQUIRED
          data IDREFS #REQUIRED
>
<!-- join -->
<!ELEMENT join EMPTY >
```

```
<!ATTLIST join
          name           ID    #REQUIRED
          left-data      IDREF #REQUIRED
          left-variable  CDATA #REQUIRED
          right-data     IDREF #REQUIRED
          right-variable CDATA #REQUIRED
>
<!-- view -->
<!ELEMENT view (result-node) >
<!ATTLIST view
          name     ID    #REQUIRED
          data     IDREF #REQUIRED
          order-by CDATA #IMPLIED
          group-by CDATA #IMPLIED
>
<!-- result node -->
<!ELEMENT result-node (result-node*) >
<!ATTLIST result-node
          type  CDATA #REQUIRED
          value CDATA #REQUIRED
>
```

Annexe B

Sources de données XML

B.1 Source de données biblio

```
<?xml version="1.0" encoding="ISO8859_1" ?>
<!-- dtd -->
<!DOCTYPE dtd SYSTEM  "biblio.dtd" >

<!-- element racine -->
<bibliographie>

<!-- auteurs -->
<auteur id="xb">
<nom>Baril</nom>
<prenom>Xavier</prenom>
<email>xavier.baril@free.fr</email>
<email>baril@lirmm.fr</email>
<web>http://xavier.baril.free.fr</web>
</auteur>

<auteur id="zb">
<nom>Bellahsène</nom>
<prenom>Zohra</prenom>
<email>bella@lirmm.fr</email>
</auteur>

<!-- publications -->
<publication id="caise2003" auteurs="xb zb">
<titre>Selection of Materialized Views: a Cost Based Approach</titre>
<type>Conférence internationale</type>
<annee>2003</annee>
<titre-livre>Conference on Advanced Information Systems (CAISE)</titre-livre>
<pages>??</pages>
</publication>

<publication id="book2003" auteurs="xb zb">
<titre>Designing and Managing an XML Warehouse</titre>
<type>Chapitre de livre</type>
<annee>2003</annee>
<titre-livre>XML Data Management</titre-livre>
<pages>Chapitre 16</pages>
```

```
<isbn>0-201-84452-4</isbn>
</publication>

<publication id="isi2001" auteurs="zb xb">
<titre>XML et les systèmes d'intégration de données</titre>
<type>Revue nationale</type>
<annee>2001</annee>
<titre-livre>Ingénierie des systèmes d'information (ISI)</titre-livre>
<pages>11-32</pages>
</publication>

<publication id="oois2001" auteurs="xb zb">
<titre>A Browser for Specifying XML Views</titre>
<type>Conférence internationale</type>
<annee>2001</annee>
<titre-livre>Object Oriented Information System (OOIS)</titre-livre>
<pages>164-174</pages>
</publication>

<publication id="oois2000" auteurs="xb zb">
<titre>A View Model for XML Documents</titre>
<type>Conférence internationale</type>
<annee>2000</annee>
<titre-livre>Object Oriented Information System (OOIS)</titre-livre>
<pages>429-441</pages>
</publication>

<!-- fin du document -->
</bibliographie>
```

B.2 DTD de la source de données biblio_lirmm

```
<?xml version="1.0" encoding="ISO8859_1" ?>

<!-- element racine -->
<!ELEMENT publications (publication*) >

<!-- publications -->
<!ELEMENT publication (titre, auteur+, type, annee, titre-livre, isbn?) >
<!ATTLIST publication
          id ID #REQUIRED
>

<!-- auteurs -->
<!ELEMENT auteur EMPTY >
<!ATTLIST auteur
          id     ID    #REQUIRED
          nom    CDATA #REQUIRED
          prenom CDATA #REQUIRED
          email  CDATA #REQUIRED
>

<!-- PCDATA -->
<!ELEMENT nom         (#PCDATA) >
```

```
<!ELEMENT prenom      (#PCDATA) >
<!ELEMENT titre       (#PCDATA) >
<!ELEMENT type        (#PCDATA) >
<!ELEMENT annee       (#PCDATA) >
<!ELEMENT titre-livre (#PCDATA) >
<!ELEMENT isbn        (#PCDATA) >
```

B.3 DTD de la source de données librairie

```
<?xml version="1.0" encoding="ISO8859_1" ?>

<!-- element racine -->
<!ELEMENT livres (livre*) >

<!-- livres -->
<!ELEMENT livre (titre, auteur+, annee, prix) >
<!ATTLIST publication
          isbn ID #REQUIRED
>

<!-- auteurs -->
<!ELEMENT auteur (nom, prenom) >

<!-- PCDATA -->
<!ELEMENT titre  (#PCDATA) >
<!ELEMENT nom    (#PCDATA) >
<!ELEMENT prenom (#PCDATA) >
<!ELEMENT annee  (#PCDATA) >
<!ELEMENT prix   (#PCDATA) >
```

Annexe C

Intégration de données avec VIMIX

C.1 Spécification du motif sp_auteurs_biblio

```
<source-pattern name="sp_auteurs_biblio" source="biblio">
  <search-axis function="children">
    <source-node reg-expression="auteur" type="element">
      <search-axis function="children">
        <source-node reg-expression="nom"
                     type="element"
                     bindto="nom">
        </source-node>
        <source-node reg-expression="prenom"
                     type="element"
                     bindto="prenom">
        </source-node>
      </search-axis>
    </source-node>
  </search-axis>
</source-pattern>
```

C.2 Spécification du motif sp_auteurs_biblio_lirmm

```
<source-pattern name="sp_auteurs_biblio_lirmm" source="biblio-lirmm">
  <search-axis function="children">
    <source-node reg-expression="publication" type="element">
      <search-axis function="children">
        <source-node reg-expression="auteur" type="element">
          <search-axis function="children">
            <source-node reg-expression="nom"
                         type="attribute"
                         bindto="nom" />
            <source-node reg-expression="prenom"
                         type="attribute"
                         bindto="prenom" />
            <source-node reg-expression="email"
                         type="attribute"
                         bindto="email" />
          </search-axis>
        </source-node>
```

```
      </search-axis>
    </source-node>
  </search-axis>
</source-pattern>
```

C.3 Spécification du motif sp_livres

```
<source-pattern name="sp_livres" source="librairie">
  <search-axis function="children">
    <source-node reg-expression="livre" type="element">
      <search-axis function="children">
        <source-node reg-expression="titre"
                     type="element"
                     bindto="titre">
        </source-node>
        <source-node reg-expression="auteur" type="element">
          <search-axis function="children">
            <source-node reg-expression="nom"
                         type="attribute"
                         bindto="auteur">
            </source-node>
          </search-axis>
        </source-node>
        <source-node reg-expression="prix"
                     type="element"
                     bindto="prix">
        </source-node>
      </search-axis>
    </source-node>
  </search-axis>
</source-pattern>
```

C.4 Spécification du fragment f_auteurs

```
<fragment name="f_auteurs" data="sp_auteurs_biblio_lirmm sp_auteurs_biblio">
  <restrictions>
    <restriction variable="nom" priority="sp_auteurs_biblio_lirmm" />
  </restrictions>
</fragment>
```

C.5 Spécification de la jointure j_livres_lirmm

```
<join name="j_livres_lirmm"
      left-data="sp_livres"
      left-variable="auteur"
      right-data="f_auteurs"
      right-variable="nom"
/>
```

Annexe D

Requêtes SQL pour la maintenance de vues VIMIX

D.1 Requêtes SQL pour la mise à jour des *mappings* d'un fragment

Si la source s est un fragment, on va procéder de la manière suivante. Tout d'abord, il faut calculer les lignes à supprimer dans la table T_s représentant le fragment et les stocker dans la table des suppressions, notée $T_{deletes}$. Ce traitement peut s'exprimer à l'aide de la requête SQL de la Figure D.1.

```
insert into table T_deletes
select * from T_s
where Fix(id) = order_child and sID in (select id from T_deletes_child) ;
```

FIG. D.1 – Requête SQL calculant les lignes à supprimer dans un fragment.

Cette requête insére dans la table des suppressions ($T_{deletes}$), les lignes de la table représentant le fragment (T_s), provenant de la source mise à jour (*child*) qui ont été supprimées. On calcule ces lignes en sélectionnant dans la table représentant le fragment (T_s) toutes les lignes qui respectent les deux conditions suivantes :

1. La ligne provient de la source qui a été mise à jour. Pour cela, on teste l'égalité entre la partie entière de l'identifiant de la ligne (`Fix(id)`) et le numéro d'ordre de la source mise à jour dans le fragment ($order_{child}$).
2. La ligne provient d'une ligne qui a été supprimée dans la source mise à jour. Pour cela, on teste que l'identifiant de la source dont provient la ligne, appartient à l'ensemble des lignes qui ont été supprimées dans la source.

Ensuite, on doit calculer les lignes à ajouter dans la table représentant le fragment. Ce sont les lignes de la table représentant la source mise à jour qui respectent les conditions et les restrictions exprimées par le fragment. Ce traitement peut s'exprimer à l'aide de la requête SQL de la Figure D.2.

```
insert into T_inserts(id, sID, variables_child)
select getID(s, sID), id, variables_child
from T_inserts_child
where sql(conditions_s) and sql(restictions_s) ;
```

FIG. D.2 – Requête SQL calculant les lignes à ajouter dans un fragment.

Cette requête insere dans la table des insertions ($T_{inserts}$), les lignes de la source qui a été mise à jour. Pour cela, la requête doit créer un nouvel identifiant pour la ligne avec la fonction $getID$. Ensuite, elle sélectionne l'identifiant et les variables de la table représentant les données insérées dans la source mise à jour ($T_{inserts_{child}}$) qui respectent les conditions et les restrictions du fragment.

Si la source du fragment qui a été mise à jour n'est pas utilisée comme source prioritaire dans une restriction, alors les modifications des données de cette source dans le fragment n'ont pas d'effet sur les données provenant des autres sources. Dans le cas contraire, il faut mettre à jour les données du fragment provenant des autres sources, pour qu'elles respectent les restrictions qui sont dépendantes de la source mise à jour (*child*). Pour cela, deux opérations sont nécessaires.

1. Il faut supprimer les données du fragment qui ne sont plus cohérentes avec les restrictions.
2. Pour chaque source du fragment (à l'exception de celle qui a été modifiée), il faut ajouter les données qui sont cohérentes avec les restrictions.

La requête SQL de la Figure D.3 permet de calculer les suppressions à effectuer lorsque la source mise à jour est utilisée dans une restriction du fragment. L'ensemble des restrictions du fragment utilisant la source mise à jour (*child*) comme source prioritaire est noté $restrictions_{child}$.

```
insert into T_deletes
select * from T_s
where not (sql(restrictions_child)) and sID not in (select id from T_child) ;
```

FIG. D.3 – Requête SQL calculant les suppressions lorsque la source mise à jour est utilisée dans une restriction.

Cette requête ajoute dans la table des suppressions ($T_{inserts}$) toutes les lignes de la table représentant le fragment (T_s), qui contiennent des données ne respectant pas les restrictions utilisant la source mise à jour ($restrictions_{child}$) et ne provienant pas de la source mise à jour.

La requête SQL de la Figure D.4 permet de calculer les ajouts à effectuer lorsque la source mise à jour est utilisée dans une restriction du fragment. Cette requête sera exécutée pour chaque source utilisée par le fragment (*otherchild*) , qui est différente de la source mise à jour (*child*).

Cette requête ajoute dans la table des insertions ($T_{inserts}$), toutes les lignes de la table de la source à mettre à jour ($T_{otherchild}$), qui respectent les restrictions et les conditions du fragment et qui ne sont pas déjà insérées. Les lignes de $T_{otherchild}$ qui sont déjà insérées sont éliminées par la condition de la

```
insert into T_inserts(id, variables_otherchild)
select getID(otherchild, sID), id, variables_otherchild
from T_otherchild
where sql(conditions_s) and sql(restrictions_s) and
id not in (select sID from T_s where Fix(id) = order_otherchild) ;
```

FIG. D.4 – Requête SQL calculant les ajouts lorsque la source mise à jour est utilisée dans une restriction.

sous-requête qui compare la partie entière de l'identifiant du fragment `Fix(id)` avec le numéro d'ordre de la source, noté $order_{otherchild}$.

D.2 Requêtes SQL pour la mise à jour des *mappings* d'une jointure

Si la source s qui a été mise à jour est une jointure, on va procéder de la manière suivante. Tout d'abord il faut calculer les lignes à supprimer de la jointure. Ces lignes sont celles qui proviennent de la source qui a été mise à jour. Ce traitement peut s'exprimer à l'aide de la requête SQL de la Figure D.5.

```
insert into table T_deletes
select * from T_s
where [lsID | rsID] in (select id from T_deletes_child) ;
```

FIG. D.5 – Requête SQL calculant les lignes à supprimer dans une jointure.

Cette requête insére dans la table des suppressions ($T_{deletes}$), les lignes de la table représentant la jointure (T_s), provenant de la source mise à jour (*child*) qui ont été supprimées. On calcule ces lignes en sélectionnant les lignes de la table représentant la jointure (T_s), qui proviennent des lignes de la table contenant les suppressions de la source. Si la source mise à jour est utilisée dans la partie gauche de la jointure, on utilisera la colonne `lsID` dans la condition de sélection, si elle est utilisée dans la partie droite, on utilisera la colonne `rsID`.

Ensuite, on doit calculer les lignes à ajouter dans la table représentant la jointure. Pour cela, on calcule la jointure entre les lignes qui ont été ajoutées dans la source mise à jour et l'autre source de données. Ce traitement peut s'exprimer à l'aide de la requête SQL de la Figure D.6.

```
insert into table T_inserts
select getID(), l.id, r.id, variables_leftdata, variables_rightdata
from [T_inserts_child | T_leftdata] l, [T_inserts_child | T_rightdata] r
where text(l.leftvariable) = text(r.rightvariable) ;
```

FIG. D.6 – Requête SQL calculant les lignes à insérer dans une jointure.

Cette requête insére dans la table des insertions ($T_{inserts}$), le résultat de la jointure entre les lignes qui ont été ajoutées dans la source mise à jour et les lignes de l'autre source. Les lignes ajoutées dans la source mise à jour sont contenues dans la table $T_{inserts_{child}}$. Si la source mise à jour constitue la partie gauche de la jointure, alors $T_{inserts_{child}}$ sera utilisée comme table notée `l`. Si la source mise à jour constitue la partie droite de la jointure, alors $T_{inserts_{child}}$ sera utilisée comme table notée `r`.

Annexe E

Interrogation de vues VIMIX

E.1 Un langage d'interrogation pour l'entrepôt

Dans cette section, nous présentons le langage que nous proposons pour interroger l'entrepôt de données. Notre langage est très proche de XPath [XPa99], qui a été défini par le W3C pour localiser des nœuds à l'intérieur d'un document XML. Cependant, notre modèle de données pour XML est différent de celui utilisé dans XPath. Notre langage d'interrogation profite des apports de notre modèle : il permet de définir facilement des expressions de navigation qui utilisent les mécanismes de partage d'éléments proposés par XML. En effet, dans notre modèle de données, les nœuds représentant des attributs `IDREF(S)` possèdent des fils, qui sont les éléments référencés par les attributs. Le modèle de données utilisé par XPath considère les références comme de simples valeurs qui n'ont pas de sémantique particulière.

XPath et le langage d'interrogation que nous proposons sont des langages de **localisation**. Ils permettent de localiser des données dans un document XML[1], mais ne permettent pas de restructurer les éléments du résultat. Nous pensons que cette fonctionnalité est inutile dans le contexte de l'interrogation de notre entrepôt de données. En effet, la définition des vues sur les sources permet déjà de restructurer les données, afin d'en fournir une vue unifiée. De plus, nous pensons qu'il existe principalement deux avantages à utiliser un langage de localisation simple pour interroger l'entrepôt de données.

1. Tout d'abord, l'exécution des requêtes sera plus performante. En effet, il est plus facile de traiter des requêtes qui interrogent seulement la structure des données de l'entrepôt, plutôt que des requêtes exprimées avec un langage plus puissant permettant de restructurer les données et de créer de nouveaux éléments. De plus, notre méthode de stockage permet de traduire efficacement des requêtes qui interrogent les données de l'entrepôt en utilisant leur structure.
2. Ensuite, l'interrogation de l'entrepôt de données est facilitée pour les utilisateurs si le langage de requête est simple. Le langage de localisation que nous proposons, comme XPath, permet

[1] Dans notre contexte, le document XML est défini comme le contenu de l'entrepôt.

d'interroger l'entrepôt en exprimant la structure des données à rechercher avec une notation proche de celles des URLs. Cette notation, basée sur la notion de chemin dans les données XML, permet aux utilisateurs d'exprimer facilement des requêtes sur l'entrepôt.

Après une présentation d'XPath, nous présenterons le langage que nous défini pour l'interrogation de l'entrepôt.

E.1.1 Le langage XPath

Nous présentons ici le fonctionnement général du langage XPath, mais on pourra se référer à la norme du langage [XPa99] pour une présentation plus complète. XPATH a été défini par le W3C principalement pour localiser les données d'une partie d'un document XML. A l'origine, il avait été défini dans un effort d'homogénéisation de la syntaxe et de la sémantique des fonction communes à XSLT [XSL99] et XPointer [XPo]. Son nom illustre l'utilisation d'une écriture de type "chemin d'accès" pour se déplacer à l'intérieur de la structure hiérarchique d'un document XML.

Fonctionnement

Dans XPath, un document XML est représenté par un arbre de nœuds. Il y a plusieurs types de nœuds, parmi lesquels les nœuds éléments, attributs et textuels. Le fonctionnement d'XPath repose sur l'utilisation de **chemins de localisation**. Un tel chemin permet de localiser des données dans document XML. Il existe deux sortes de chemin : les chemins relatifs et les chemins absolus.

Un **chemin relatif** est constitué d'un ensemble d'étapes séparées par le caractère `/`. Les étapes sont assemblées de gauche à droite. Chacune à leur tour, elles sélectionnent un ensemble de nœuds à partir d'un **nœud contextuel**. La séquence initiale d'étapes est ensuite composée avec l'étape suivante. Pour cela, on applique deux règles.

1. La séquence initiale sélectionne un ensemble de nœuds qui seront utilisés un à un comme les nœuds contextuels de l'étape suivante.
2. Les ensembles de nœuds sélectionnés par l'étape suivante sont ensuite réunis pour constituer le résultat.

Un **chemin absolu** est constitué du caractère `/` suivi éventuellement d'un chemin relatif. Le caractère `/` désigne l'élément racine du document sur lequel est appliqué le chemin. Lorsqu'il est suivi d'un chemin relatif, le nœud contextuel de la première étape du chemin relatif est l'élément racine du document.

Un chemin de localisation est constitué d'un ensemble d'**étapes de localisation**. Chaque étape de localisation s'exécute en trois temps à partir du nœud contextuel. Pour cela, une étape de localisation spécifie trois propriétés.

1. Un **axe** qui spécifie le type de relation arborescente (le sens de la recherche) entre le nœud contextuel et les nœuds à localiser.
2. Un **nœud test** qui spécifie le type et le nom des nœud que la recherche doit localiser.
3. Zéro ou plusieurs **prédicats** qui permettent de filtrer l'ensemble des nœuds obtenus.

XPATH propose de nombreux axes permettant de spécifier une relation arborescente. Nous présentons ici ceux qui nous semble les plus utiles.

- `child` : l'axe contient les enfants directs du nœud contextuel.
- `descendant` : l'axe contient les descendants du nœud contextuel. Un descendant est un enfant à un niveau quelconque (enfant, petit-enfant, ...).
- `parent` : l'axe contient le parent du nœud contextuel, s'il existe.
- `ancestor` : l'axe contient les ancêtres du nœud contextuel. Un ancêtre est un parent à un niveau quelconque (parent, grand-parent, ...).
- `self` : l'axe contient le nœud contextuel.
- `descendant-or-self` : l'axe contient les descendants du nœud contextuel et le nœud contextuel lui-même.
- `attribute` : l'axe contient les attributs du nœud contextuel.
- ...

Le nœud de test permet de sélectionner les nœuds de l'axe selon un critère. Ce critère peut être un nom d'élément ou d'attribut, une expression régulière ou une fonction (`text()`, `attribute()`, `node()`, ...).

Syntaxe et cas d'utilisation de XPath

Une étape de localisation est de la forme `axis::node[predicats]` et la présence de prédicat(s) n'est pas obligatoire. Dans un chemin de localisation, les différentes étapes sont séparées par `/`.

Nous allons illustrer le fonctionnement et la syntaxe d'XPath à travers quelques exemples d'utilisation. Pour chaque expression XPath, nous donnons la sémantique correspondante en français.

- `child::patient`
  Sélectionne tous les éléments `patient` qui sont les enfants du nœud contextuel.
- `child::patient/attribute::nom`
  Sélectionne tous les attributs `nom` des éléments `patient` qui sont les enfants du nœud contextuel.
- `child::patient[attribute::nom='ZOLA']`
  Sélectionne tous les éléments `patient` qui sont les enfants du nœud contextuel et qui ont un attribut `nom` ayant pour valeur `'ZOLA'`.
- `/descendant-or-self::patient[attribute::nss]`
  Chemin absolu qui sélectionne tous les éléments `patient` qui sont des descendants de la racine et qui ont un attribut `nom`.

Syntaxe abrégée

Il existe une syntaxe abrégée pour écrire les chemins de localisation XPath. Cette syntaxe permet d'écrire les chemins de localisation les plus utilisés dans un style proche de celui des URLs. Voici les règles d'abréviation les plus importantes :

- `child::` peut être omis,
- `attribute::` peut être abrégé en `@`,
- `/descendant-or-self::node()/` peut être abrégé en `//`.

Nous allons maintenant réécrire les expressions XPath précédentes en utilisant cette notation abrégée :

- `child::patient` devient `patient`,
- `child::patient/attribute::nom` devient `patient/@nom`,
- `child::patient[attribute::nom='ZOLA']` devient `patient[@nom='Zola']`,
- `/descendant-or-self::patient[attribute::nss]` devient `//patient[@nss]`.

L'utilisation de cette syntaxe abrégée permet d'écrire de nombreuses expressions XPath de manière intuitive, en utilisant une notation proche de celles des URLs. Cette syntaxe permet donc de faciliter l'écriture de requêtes XPath aux utilisateurs.

E.1.2 Langage d'interrogation de l'entrepôt défini avec VIMIX

Le langage d'interrogation que nous proposons possède un fonctionnement similaire à celui d'XPath. Une requête est composée de chemins qui permettent de spécifier la structure des données à rechercher dans l'entrepôt. Cependant, le modèle de données que nous utilisons est différent. Les nœuds représentant les attributs de type `IDREF(S)` peuvent avoir un ou plusieurs fils, représentant les éléments qui sont référencés. De plus, nous proposons uniquement une syntaxe abrégée de notre langage. Nous avons limité les axes de recherche disponibles pour naviguer dans le graphe représentant les données. Ces choix permettent de faciliter l'écriture et le traitement des requêtes.

XPath permet d'exprimer des requêtes sur des documents XML. Dans le contexte d'un entrepôt, ce sont les données stockées dans l'entrepôt qui sont interrogées. Le schéma médiateur de l'entrepôt permet de présenter une vue unifiée des données des sources qui sont intégrées. L'interrogation de l'entrepôt se fait en utilisant ce schéma médiateur. De ce fait, on peut considérer l'entrepôt comme un document, dont la structure est définie par le schéma médiateur. Notre langage d'interrogation permet d'exprimer des requêtes sur l'entrepôt en le considérant comme un seul document. Ce document est validé par la DTD générée en utilisant la spécification des vues de l'entrepôt. La construction du schéma médiateur a été traitée dans la partie concernant la spécification d'un système d'intégration de données XML avec VIMIX (4.2.2). Dans la suite de cette section, nous utiliserons le terme document pour désigner le contenu de l'entrepôt.

La Figure E.1 décrit la grammaire du langage que nous avons défini pour interroger l'entrepôt. Une requête est constituée d'une liste de chemins de longueur 1, exprimant les données à rechercher dans

1. **query** : **path**+
2. **path** : (('/'|'//')(elem-name|'text()') | ('/@'|'//@')att-name) **filter***
3. **filter** : '[' ('text()='value | **path**) ']'

FIG. E.1 – Grammaire décrivant le langage d'interrogation de l'entrepôt.

le document. La concaténation de tous ces chemins constitue un chemin (de longueur égale au nombre de chemins de longeur 1) permettant d'interroger le contenu de l'entrepôt. Le résultat d'une requête est la liste des nœuds du document désignés par le chemin exprimé par cette requête.

Comme dans XPath, le nœud contextuel utilisé pour l'évaluation du premier chemin (de longueur 1) de la requête, est l'élément racine du document. Les autres chemins utiliseront comme nœuds contextuels, les nœuds désignés par le chemin composé des chemins précédents dans la liste.

Un chemin de longueur 1 est constitué d'un axe de recherche et d'un élément à rechercher dans le document. La forme et la sémantique des chemins que l'on peut exprimer avec notre langage sont présentés dans le Tableau E.1.

Chemin	**Sémantique**
`/element-name`	Recherche les nœuds fils (de type élément) du nœud contextuel dont le nom est `element-name`.
`//element-name`	Recherche les nœuds descendants (de type élément) du nœud contextuel dont le nom est `element-name`. Cette recherche s'effectue en utilisant les liens de référence.
`/@attribute-name`	Recherche les nœuds fils (de type attribut) du nœud contextuel dont le nom est `attribute-name`.
`//@attribute-name`	Recherche les nœuds descendants (de type attribut) du nœud contextuel dont le nom est `attribute-name`. Cette recherche s'effectue en utilisant les liens de référence.
`/text()`	Recherche les nœuds fils (de type texte) du nœud contextuel.
`//text()`	Recherche les nœuds descendants (de type texte) du nœud contextuel, en utilisant les liens de référence.

TAB. E.1 – Sémantique des chemins de longueur 1 du langage d'interrogation.

Un chemin peut éventuellement être complété par une liste de conditions. Ces conditions permettent de filtrer les données du résultat de la requête. Elles devront être vérifiées par les nœuds désignés par le chemin sur lesquels elles sont définies. Nous proposons deux sortes de conditions.

1. La comparaison d'un des nœuds fils de type texte du nœud (devant vérifier la condition) avec une valeur. Si le nœud désigné par le chemin n'a pas de fils texte, alors la condition ne sera pas

vérifiée.

2. L'existance d'un chemin à partir du nœud devant vérifier la condition. Ce chemin est composé de la même manière que les chemins constituant la requête.

E.2 Traitement d'une requête

Dans cette section, nous présentons les algorithmes qui permettent d'exécuter les requêtes XML sur l'entrepôt de données. Les données XML des sources et les *mappings* des vues sont stockées dans un SGBD relationnel. Il est nécessaire de réécrire en SQL les requêtes posées sur le schéma médiateur de l'entrepôt pour les exécuter. Afin de simplifier la présentation des algorithmes nous n'avons pas pas pris en compte les filtres que nous avons présentés dans le langage d'interrogation. Cependant, ces filtres peuvent être rajoutés dans les requêtes SQL construites, grâce à l'utilisation de la clause `exists`. Les algorithmes qui permettent de générer les filtres dans les requêtes SQL, sont similaires à ceux que nous allons présenter pour le traitement des requêtes.

La fonction `execute-query` décrite par l'algorithme 13 permet d'exécuter une requête XML posée à l'entrepôt. Lorsqu'une requête est posée à l'entrepôt, son résultat peut contenir deux types de données.

1. Des données construites par les vues.
2. Des données brutes extraites des sources.

Les vues permettent de restructurer les données des sources, en créant de nouveaux éléments ou attributs. Ces nouveaux éléments ne sont pas stockés dans les données de l'entrepôt, ils sont générés lors de la construction du résultat de la vue. L'avantage de cette technique, c'est qu'elle **exploite la structure des données des vues pour répondre aux requêtes**. En effet, lorsqu'une requête est posée au système, il n'est pas nécessaire de parcourir les données de la vue pour calculer la réponse. Il suffit de rechercher le chemin exprimé par la requête dans le schéma médiateur de l'entrepôt.

La fonction `query-execution` accepte en paramètre la requête XML posée au système ($query_{xml}$). Pour construire son résultat, il faut d'abord construire le graphe de données qui contiendra les données du résultat (G_{root}). Ensuite, deux étapes sont nécessaires.

1. Il faut reconstruire les données du résultat qui correspondent à des données créées par les vues.
2. Il faut reconstruire les données du résultat qui correspondent à des données qui proviennent des sources.

Algorithme 13: `query-execution(`$query_{xml}$`)`

Résultat : Renvoie le résultat d'une requête XML
$root \leftarrow$ `new-document('query-result')` ;
// Etape 1 : construction des données des vues ;
$P \leftarrow$ ensemble de chemins du schéma médiateur pouvant réponde à $query_{xml}$;
pour chaque $p \in P$ **faire**
  $v \leftarrow$ vue ayant généré p ;
  n nœud du résultat de v correspondant à l'extrémité de p ;
  $gb_n \leftarrow gb_v \cap variables_n$;
  $q_{sql} \leftarrow$ `rewrite-schema1(p)` ;
  `view-construction(`n`,` gb_n`, query(`q_{sql}`),` $root$`)` ;
// Etape 2 : construction des données des sources ;
$Q_{xml} \leftarrow$ ensemble des sous-requêtes de $query_{xml}$ pouvant être satisfaites par des données de l'entrepôt ;
pour chaque $subquery_{xml} \in Q_{xml}$ **faire**
  $subquery_{sql} \leftarrow$ `rewrite-schema2(`$subquery_{xml}$`)` ;
  $q_{sql} \leftarrow$ `rewrite-data(`$query - subquery_{xml}$`,` $subquery_{sql}$`)` ;
  $IDs \leftarrow$ `query(`q_{sql}`)` ;
  pour chaque $nodeID \in IDs$ **faire**
    `node-reconstruction(`$nodeID$`,` $root$`)` ;

La première étape est de construire les données qui correspondent à des données créées par les vues. Pour cela, il faut rechercher l'ensemble des chemins du schéma médiateur (P) qui permettent de répondre à la requête $query_{xml}$.

Ensuite, pour chacun de ces chemins, il faut déterminer la vue qui a générée ce chemin dans le schéma médiateur. Cette étape permet de **localiser les données** qui permettrons de répondre à la requête. Le chemin spécifié par la requête peut correspondre seulement à une partie de la vue. Pour cette raison, il faut déterminer dans l'arbre spécifiant la structure du résultat de la vue, le nœud (n) qui correspont à l'extrémité du chemin exprimé par la requête (p). Ce nœud sera la racine du sous-arbre spécifiant les données de la vue à reconstruire.

Ensuite, il faut calculer les niveaux de regroupement du sous-arbre correspondant à la requête. Pour cela, on fait l'intersection de la liste des variables indiquant les niveaux de regroupement de la vue (gb_v) et de la liste des variables utilisées par le sous-arbre ($variables_n$).

Enfin, on construit la requête SQL qui permet d'obtenir les *mappings* qui correspondent au sous-arbre de la spécification de la vue. La construction de cette requête est effectée par la fonction `rewrite-schema1` dont le fonctionnement est expliquée dans la sous-section suivante (E.2.1). Le résultat de cette requête est utilisé pour reconstruire les nœuds des données de la vue, en utilisant la fonction `view-construction` qui a été présentée plus haut (algorithme 8).

La deuxième étape est de construire les données qui correspondent aux données brutes des sources. Pour cela, il faut rechercher l'ensemble des chemins du schéma médiateur qui permettent de répondre à la requête. Cet ensemble, qui permet de construire des sous-requêtes XML (Q_{xml}), est constitué par les chemins suivants.

- Tous les chemins du schéma médiateur qui contiennent des sous-chemins de la requête et qui sont terminés par l'axe de recherche `//`. Par exemple, si la requête XML à traiter est `//adresse//ville`, les sous chemins existants dans le schéma médiateur sont : `//` et tous les chemins qui contiennent l'élément `adresse`.
- Tous les chemins du schéma médiateur qui contiennent des sous-chemins de la requête et qui sont terminés par une variable. Si l'on considère l'exemple de requête précédent et que le schéma médiateur contient un chemin `/view/personne/adresse/ville/variable:adresse`, alors les *mappings* de ce chemin permettrons de répondre à la requête.

Ensuite, pour chaque requête de cet ensemble ($subquery_{xml}$), on construit la sous-requête SQL correspondante ($subquery_{sql}$). Cette requête SQL permet de trouver tous les *mappings* qui contiennent des éléments pouvant répondre à la requête de départ. Elle est construite avec la fonction `rewrite-schema2` dont le fonctionnement est expliqué dans la sous-section suivante (E.2.1).

Ensuite, le reste de la requête ($query_{xml} - subquery_{xml}$) est réécrit en une requête SQL qui permet de rechercher les données stockées dans l'entrepôt qui peuvent répondre au résultat de la requête. Pour cela, on utilise la fonction `rewrite-data` qui permet de réécrire une requête SQL sur les données des sources. La réécriture s'effectue récursivement, en utilisant les règles de transformation qui sont présentées dans la sous-section E.2.2.

E.2.1 Réécriture d'une requête sur le schéma médiateur de l'entrepôt

La fonction `rewrite-schema1` construit une requête SQL qui renvoie les *mappings* permettant de construire le résultat d'une vue. Elle accepte en paramètre un chemin (p) du schéma médiateur de l'entrepôt. Ce chemin est utilisé pour déterminer les variables qui sont utilisées par la partie de la vue à construire. Si on note n le nœud du résultat de la vue v ayant généré l'extrémité du chemin p, alors la requête SQL construite par la fonction `rewrite-schema1` sera :
`select` $variables_n$ `from` T_v
avec $variables_n$ les variables utilisées dans le sous-arbre ayant pour racine n et T_v la table contenant les *mappings* de la vue v.

Exemple Considérons la vue `vue-patients` (dont le schéma est celui décrit par la Figure 5.13) et le chemin dans le schéma médiateur désignant les opérations des patients.
`rewrite-schema1(/vue-patients/patient/operation)='select` $date$, $objet$ `from` $T_{vue-patients}$'
Cette requête permettra de construire la partie de la vue qui contient les éléments `operation`, à partir des *mappings* des variables *date* et *objet*.

La fonction `rewrite-schema2` construit une requête SQL qui renvoie tous les *mappings* contenant les données d'une sous-requête XML ($subquery_{xml}$). Pour cela, elle génére tous les chemins du schéma médiateur pouvant satisfaire $subquery_{xml}$ et qui contiennent une variable. Ensuite, elle construit la requête SQL faisant l'union de toutes les variables. La génération de tous les chemins du schéma médiateur permet de localiser les *mappings* pouvant répondre à la requête.

Exemple Considérons la vue `vue-patients` et la sous-requête XML recherchant les descendants des éléments `operation` : `/vue-patient/patient/operation//`. Il y a deux chemins qui permettent de trouver des variables qui satisfassent cette sous-requête :

- `/vue-patient/patient/operation/date/variable:date`
- `/vue-patient/patient/operation/objet/variable:objet`

La requête construite fera donc l'union de ces variables :
`rewrite-schema2(/vue-patient/patient/operation//)=`
`(select` *date* `from` $T_{vue-patients}$`) union (select` *objet* `from` $T_{vue-patients}$`)`

Cas particulier Lorsque la sous-requête XML passée en paramètre à la fonction `rewrite-schema1` est `//`, tous les *mappings* des vues de l'entrepôt peuvent répondre à la requête. Plutôt que de faire l'union de toutes les variables des tables qui contiennent ces *mappings*, on renvoie la requête SQL suivante :
`select nodeID from XmlNode`
qui renvoient les idendifiants de tous les nœuds stockés dans l'entrepôt.

E.2.2 Réécriture d'une requête sur les données XML de l'entrepôt

La fonction `rewrite-data` construit une requête SQL qui renvoie tous les *mappings* contenant les données d'une sous-requête XML. Elle accepte les paramètres suivants.

- $subquery_{xml}$ une sous-requête XML à réécrire.
- $subquery_{sql}$ est une sous-requête SQL qui renvoie les identifiants des nœuds candidats. Ce sont ces nœuds candidats que la requête $subquery_{xml}$ doit satisfaire.

Pour construire la requête SQL, on utilise les **règles de transformation** qui sont décrites dans le Tableau E.2. La construction s'effectue en décomposant la requête XML passée en paramètre. Pour chaque étape de cette requête, on applique la règle de transformation correspondante, en utilisant la requête SQL qui correspond au chemin précédent. Cette requête est notée *subquery* dans les règles de transformation.

La règle de l'étape */element*, permet de construire une requête SQL recherchant les éléments fils ayant un nom donné. La jointure entre les tables `Element` et `XmlNode` permet de trouver tous les nœuds de type élément qui ont le nom recherché. La jointure avec la table `Children` permet de ne sélectionner que les nœuds dont le père appartient à la sous-requête.

La règle de l'étape `//`*element*, permet de construire une requête SQL recherchant les éléments descendants ayant un nom donné. La jointure entre les tables `Element` et `XmlNode` permet de trouver tous les nœuds de type élément qui ont le nom recherché. La jointure avec la table `Descendants` permet de ne sélectionner que les nœuds qui sont des descendants des nœuds renvoyés par la sous-requête.

La règle de l'étape `/`*attribute*, permet de construire une requête SQL recherchant l'attribut ayant un nom donné. La jointure entre les tables `Attribute` et `XmlNode` permet de trouver tous les nœuds de type attribut qui ont le nom recherché. La jointure avec la table `Children` permet de ne sélectionner que les nœuds dont le père appartient à la sous-requête.

La règle de l'étape `//`*attribute*, permet de construire une requête SQL recherchant les attributs des descendants ayant un nom donné. La jointure entre les tables `Attribute` et `XmlNode` permet de trouver tous les nœuds de type attribut qui ont le nom recherché. La jointure avec la table `Descendant` permet de ne sélectionner que les nœuds qui sont des descendants de ceux renvoyés par la sous-requête.

La règle de de l'étape `/text()`, permet de construire une requête SQL recherchant les fils de type texte. Cette requête fait l'union de deux sous-requêtes, exprimant la recherche de fils de type texte pour un élément ou un attribut. La première sous-requête recherche tous les nœuds de type texte ayant pour père les nœuds de la sous-requête. Dans notre schéma générique, les attributs ont la valeur de leur fils texte stockés dans le nœud. C'est pour cela que la seconde sous-requête renvoie les nœuds attributs de la sous-requête.

La règle de l'étape `//text()`, permet de construire une requête SQL recherchant les descendants de type texte. Pour les même raisons que précédemment, elle est composée de l'union de deux sous-requêtes. De plus, la première sous-requête utilise la jointure entre les tables `XmlNode` et `Descendants` pour calculer les descendants des nœuds de la sous-requête. Ces descendants peuvent être de type attribut, car une ligne représentant un attribut contient également son fils texte.

Etape	**Règle de transformation**

/*element*	`select xn.nodeID from Element e, XmlNode xn, Children c`
	`where e.name = '`*element*`' and`
	`e.elemID = xn.elemID and xn.nodeID = c.childID and`
	`c.fatherID in (`*subquery*`)`

//*element*	`select xn.nodeID from Element e, XmlNode xn, Descendants d`
	`where e.name = '`*element*`' and`
	`e.elemID = xn.elemID and xn.nodeID = d.childID and`
	`d.fatherID in (`*subquery*`)`

/*attribute*	`select xn.nodeID from Attribute a, XmlNode xn, Children c`
	`where a.name = '`*attribute*`' and`
	`a.attID = xn.attID and xn.nodeID = c.childID and`
	`c.fatherID in (`*subquery*`)`

//*attribute*	`select xn.nodeID from Attribute a, XmlNode xn, Descendants d`
	`where a.name = '`*attribute*`' and`
	`a.attID = xn.attID and xn.nodeID = d.childID and`
	`d.fatherID in (`*subquery*`)`

`/text()`	`(select xn.nodeID from XmlNode xn, Children c`
	`where xn.elemID is NULL and xn.attID is NULL and`
	`xn.nodeID = c.childID and c.fatherID in (`*subquery*`))`
	`union`
	`(select s.nodeID from` *subquery* `s, XmlNode xn`
	`where s.nodeID = xn.nodeID and xn.attID is not NULL)`

`//text()`	`(select xn.nodeID from XmlNode xn, Descendants d`
	`where xn.elemID is NULL and`
	`xn.nodeID = d.childID and d.fatherID in (`*subquery*`))`
	`union`
	`(select s.nodeID from` *subquery* `s, XmlNode xn`
	`where s.nodeID = xn.nodeID and xn.attID is not NULL)`

TAB. E.2 – Règles de transformation pour la réécriture de requêtes.

Exemple Considérons la requête `//patient/@nom` qui recherche les attributs `nom` des éléments `patient`, quelle que soit leur position dans le schéma médiateur de l'entrepôt. La réécriture de cette requête sur le schéma médiateur de l'entrepôt, nécessitera de considérer les données des sources qui contiennent la requête `//patient/@nom`. La figure E.2 illustre l'exemple de la réécriture de cette requête avec la fonction `rewrite-data`.

```
rewrite-data(//patient/@nom, 'select * from nodeID')=
'select xn2.nodeID
from Attribute a2, XmlNode xn2, Children c2
where
a2.name = 'nom' and a2.attID = xn2.attID and xn2.nodeID = c2.childID and
c2.fatherID in (
select xn1.nodeID
from Element e1, XmlNode xn1, Descendant d1
where
e1.name = 'patient' and e1.elemID = xn1.elemID and xn1.nodeID = d1.childID and
d1.fatherID in (select nodeID from XmlNode))'
```

FIG. E.2 – Réécriture de la requête `//patient/@nom`.

Résumé Les systèmes d'intégration de données permettent d'accéder de manière unifiée à différentes sources de données, sans se soucier de leur localisation ni de leur format. Ces systèmes reposent généralement sur des modèles de vues qui permettent de restructurer les données afin de les intégrer dans un schéma médiateur. Dans cette thèse, nous proposons un modèle de vues pour intégrer des sources de données XML. Ce modèle, appelé VIMIX (VIew Model for Integration of XML sources), permet de spécifier des opérations d'union, de jointure et de filtrage pour restructurer des sources de données XML. Des opérations d'agrégation basées sur l'algèbre relationnelle sont proposées pour spécifier le résultat de la vue. Cette spécification permet de générer un schéma sous forme de DTD, utilisé pour construire le schéma médiateur à partir d'une collection de vues. De plus, nous avons proposé une méthode de stockage pour matérialiser les vues VIMIX. Cette méthode utilise un SGBD relationnel pour construire un entrepôt dont les données peuvent être maintenues incrémentalement. Enfin, nous avons implémenté notre modèle de vues dans le système DAWAX (DAtaWArehouse for XML).

Mots-clés Intégration de données, modèle de vues, XML, vues matérialisées, entrepôt de données XML.

Abstract Data integration systems allow a unified view of several sources without considering data location or format. Usually, these systems are based on view models allowing data restructration for their integration in a mediated schema. In this thesis, we have proposed a view model for integration of XML sources. This model, that we called VIMIX (View Model for Integration of XML sources), allows to specify union, join and filtering operations to restructure XML sources. We have also proposed aggregation operations based on relational algebra to specify the view result. This specification allows generating schema (DTD), which is used for mediated schema construction defined as a collection of view. Furthermore, we have proposed a storage method to materialize VIMIX views. This method uses a relational SGBD for data warehouse construction. Data maintenance can be done in an incremental way. Finally, we have implemented our model with DAWAX (Data Warehouse for XML).

Keywords Data integration, view model, XML, materialized views, XML Data warehousing.