Fédération de données semi-structurées avec XML

Tuyet-Tram Dang-Ngoc

No. d'ordre : _____

Université de Versailles Saint-Quentin-en-Yvelines

THÈSE

pour obtenir le grade de
docteur

discipline : **Informatique**

présentée et soutenue publiquement

par

Tuyêt Trâm DANG NGOC

le 10 juin 2003

sur le sujet

Fédération de données semi-structurées avec XML

JURY

Monsieur	**Georges Gardarin**	*Directeur de thèse*
Madame	**Anne Doucet**	*Rapporteur*
Monsieur	**Patrick Valduriez**	*Rapporteur*
Monsieur	**Michel Adiba**	*Examinateur*
Monsieur	**Hubert Naacke**	*Examinateur*
Monsieur	**Lars Smit**	*Examinateur*

À mes parents qui m'ont tant donné pour faire de moi ce que je suis.
À mes frères François et Frédéric mes complices de toujours.
À Rémy qui d'une flamme nouvelle a éclairé ma vie.
Et à tous pour leur confiance, leur affection et leur amour.

Remerciements

Je tiens à remercier tous ceux qui ont contribué directement ou indirectement à l'aboutissement de cette thèse.

- Georges Gardarin mon directeur de thèse pour m'avoir accueilli dans son équipe et m'avoir accordé sa confiance durant toutes ces années de thèse ;
- Anne Doucet et Patrick Valduriez pour avoir accepté la lourde tâche d'être les rapporteurs de cette thèse ;
- Michel Adiba qui a bien voulu faire partie du jury ;
- Hubert Naacke dont les travaux m'ont beaucoup inspiré, et qui a bien voulu faire partie du jury ;
- Lars Smit qui a si bien su m'aider à ménager mon temps de thèse lorsque le travail de développement m'accaparait. Au delà de sa compétence technique et de toute l'aide et les conseils qu'il a su m'apporter, je conserve surtout son amitié ;

Je remercie aussi de tout cœur :

- les joyeux stagiaires, thésards et anciens thésards du PRiSM qui m'ont accompagné dans cette longue épreuve et avec qui j'ai eu des discussions très intéressantes, en particulier Catherine Blirando, Karine Brifault, Michel Cavaille, Tatiana Chan, Ikram Chennouf, Benjamin Cohen pour ses katas, François Galéa, Mourad Gueroui, Huaizhong Kou pour avoir égayé le bureau de sa bonne humeur et ses excellents cours d'écriture chinoise, Mathieu Le-Coz pour ses jolies interfaces graphiques, Laurent Nemirovski, Laurent Perato pour tous les combats de kung-fu qu'on a pu faire, Fei Sha, Tao Wan, Fei Wu, Lilan Wu et Xiaohui Xue ;
- Les stagiaires enthousiastes Clément Jamard et Nicolas Travers fraichement débarqués dans le monde de la médiation de données semi-structurées avec leur bonne humeur et leur efficacité ;
- les membres permanents du laboratoire PRiSM et de l'ISTY et en particulier Annick Baffert, Christiane Boucher, Michelle et Jean-Pierre Claudé, Chantal Ducoin, Denise Guiavarch, Jean-Louis Jammier et Isabelle Pendezec, pour leur compétence et leur gentillesse ;
- toute l'équipe de la société Osis où j'ai passé la première partie de ma thèse, en particulier Claude Campanaro pour son soutien et sa gentillesse, Lloyd Dupont pour sa bonne humeur communicative et son esprit curieux, Bernard Hugueney pour sa motivation si contagieuse, Dorothée Touboul pour ses encouragements et Alban Vuillier pour sa gentillesse, ses compétences techniques et son aide si précieuse dans le projet MIROWEB ;
- toute l'équipe de la société e-XMLMedia où j'ai effectué la deuxième partie de ma thèse et en particulier Bénédicte Baral, Olivier Berly, Périne Boruchowitsch, Zhangyun Lei, Marie-Félix Ruiz, Yucheng Sha et Véronique Smahi. *Marie, Périne, Lars, et Olivier, sans vous, j'aurais eu tant de mal à concilier le rythme éprouvant d'un travail alternant entre thèse et entreprise. Merci pour votre bonne humeur et votre amitié* ;
- l'équipe du CSI qui en plus de Rémy m'a donné amitié, encouragement et soutien, en

particulier Thierry Caillet, Nicolas Courtay, Pierre David, François Laforge, et Bénédicte Sapin ;

- tous les amis que j'ai pu rencontrer tout au long de ma scolarité à l'UVSQ qui m'a menée jusqu'à cette thèse : Laurent Capitaine, Nicolas Leclerc, Seindil Loganadane, Thierry Michau, Sahag Kanayan, Erwan Prioul, Sébastien Provost, Fatma Sahraoui, Éric Szabo et Pascal Varniol ;
- les étudiants du cursus info et de l'ISTY qui m'ont fait comprendre la valeur de l'enseignement et m'ont amenée à faire toujours mieux.
- et enfin du fond du cœur mes proches et mes nouveaux proches : mes parents, Frédéric, François, Rémy, Solange, Danièle, René, Xavier, Sylvie, Clément et Mathis.

Et tous les autres à qui j'adresse toute ma gratitude.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Motivation

La diversité des sources de données

L'évolution constante en matière de réseaux et de bases de données ces trente dernières années a mené à une demande toujours croissante d'accès rapides à une large quantité d'informations variées.

Certaines de ces informations sont enregistrées dans des bases de données traditionnelles et accessibles par des langages de requête très puissants, d'autres sont simplement stockées dans des systèmes de fichiers ou de simples tableurs, d'autres encore sont les résultats d'applications plus ou moins complexes, enfin, certaines sont les données changeantes et arborescentes des pages du *web*.

La diversité de ces sources de données conduit à des modes de consultation qui peuvent être très différents. Ainsi, une base de données relationnelles sera interrogée par l'intermédiaire d'une requête SQL, une page Web sera consultée par une adresse web (URL) particulière, et un tableur par une formule spécifique. Une telle variété de sources implique diverses façons de les interroger -c'est-à-dire de formuler une *requête*- mais aussi plusieurs manières pour la source de présenter un *résultat*.

Pour obtenir un résultat, la requête fait appel aux *méthodes d'accès* des sources de données. Ces méthodes d'accès peuvent se faire localement par l'intermédiaire d'une interface spécifique ou d'une interface de programmation, ou de manière distante *via* des protocoles de communication. L'apparition de protocoles comme ODBC, JDBC ou IIOP simplifient considérablement l'accès à nombre de ces sources de données. L'émergence d'architectures de médiation « trois-tiers » permet de plus en plus de regrouper l'accès à différentes sources par l'intermédiaire d'une interface unique.

Un nouveau type de données à prendre en compte : les données semi-structurées

Jusqu'à l'avènement du langage XML, le monde de l'information se divisait en deux parties complémentaires. D'un côté, il y avait le monde des bases de données traditionnelles (SGBD relationnels, et SGBD objet), de l'autre, le monde de l'EDI permettant l'échange des informations et des présentations.

Le monde des bases de données traditionnelles tire sa puissance de la régularité des structures des objets qu'il manipule, ce qui permet ainsi des langages de requête très puissants. Il ne peut par contre pas travailler sur des données dont le schéma n'est pas fixé *a priori*. Enfin, il nécessite l'ajout de composants supplémentaires quand au format et à la présentation des données.

La place sans cesse croissante de données semi-structurées dont le standard dominant est le format XML a par ailleurs conduit à définir des systèmes de base de données adaptés pour ce type de données.

XML est un langage permettant de représenter des données semi-structurées sous la forme d'éléments balisés imbriqués. Un document XML est un document textuel où les valeurs de données sont encadrées par la désignation de la donnée (ex. `<adresse>5, rue de Montreuil</adresse>`). Ainsi, le schéma est en partie défini dans les données elles-mêmes (données auto-descriptives).

La grande richesse de XML tient aussi de tous les outils et standards qui se sont développés autour de ce format. Ainsi, malgré la structure intrinsèque du schéma, une structure générale de schéma peut-être toutefois associée (XML-Schema). Un composant de présentation (feuille XSL) permet de gérer le formatage des documents et enfin des langages de requête puissants (XQuery) ont été spécifiés afin de permettre l'interrogation efficace des données.

L'émergence et la standardisation du langage XML et de composants associés pour l'échange de documents ont permis de fusionner le monde documentaire et le monde des SGBD en intégrant les fonctionnalités utiles de chacun d'eux.

1.2 Problématique

Le concept de données semi-structurées est nouveau dans le monde des bases de données, et certains parlent même de *révolution XML*.

Pour prendre en compte ce nouveau type de donnée, il faut réinventer les mécanismes et les algorithmes utilisés traditionnellement dans les bases de données.

1.2.1 Requête sur données semi-structurées

Évaluer une requête sur des données semi-structurées implique de naviguer à travers la structure en examinant à la fois les valeurs des éléments et le nom auto-descriptif de l'élément tout au long du parcours.

Plusieurs langages de requêtes sur les données semi-structurées (XML-QL, XQL) ont été proposés avant d'aboutir au langage de requête XQuery. Celui-ci est nouveau, et peu d'implémentations existent. L'implémentation de l'évaluation de requête est donc rendu complexe et peu sûre à cause de la jeunesse de ce langage.

1.2.2 Médiation de données semi-structurées

Les données sont de plus en plus disséminées sur les réseaux. Le type de ces données peut être varié (données textuelles, relationnelles, multimédia, semi-structurées) et leur système de stockage très différents (système de fichiers, SGBD, applications). Il faut offrir un système de gestion intégrant des sources de données hétérogènes tout en assurant la transparence à la distribution et à l'hétérogénéité.

L'évaluation de requêtes dans un système distribué hétérogène présente des aspects difficiles. Du fait de la localisation des données, une requête doit être divisée en sous-requêtes tenant compte de la localisation des sources et de leurs capacités. Celles-ci sont ensuite envoyées en parallèle ou en série sur les sites avec tous les problèmes d'ordonnancement et de synchronisation qui en découlent. Les problèmes concernant la médiation de données ont fait l'objet de plusieurs études. Parmi les difficultés dégagées par la conception d'un composant de médiation on peut citer principalement :

- comment intégrer des données de structures et de natures fondamentalement différentes ;
- comment décomposer une requête faisant intervenir plusieurs sources en des requêtes spécifiques à ces sources, puis savoir recomposer le résultat ;
- comment optimiser l'évaluation des requêtes dans un tel contexte distribué ;
- que faire dans le cas de sources aux possibilités d'interrogation très limitées (ex. système de fichiers).

La médiation des données semi-structurées est soumise aux problèmes inhérents à tout médiateur, mais il faut de plus tenir compte du fait que les schémas échangés peuvent être fortement arborescents et sont susceptibles d'évoluer, ou encore ne pas représenter le schéma général d'une collection de documents.

1.2.3 Algèbre d'interrogation de données semi-structurées

Dans le cas de données semi-structurées, il s'agit de concevoir une algèbre avec une assez grande puissance d'interrogation pour tirer parti de toutes les propriétés des données semi-structurées. Pour cela, l'algèbre doit à la fois se contenter du peu de typage de certains documents semi-structurés, et aussi pouvoir gérer le cas échéant des structures strictes basées sur des schémas. Enfin, l'algèbre doit pouvoir permettre des optimisations efficaces en donnant des règles d'équivalences.

Du fait de la structure arborescente des données semi-structurées, des opérations de manipulation d'arborescence (navigation, comparaison d'arbres) sont nécessaires. Or de telles opérations sont complexes et nécessitent souvent des parcours coûteux à travers des sous-arbres.

1.2.4 Modèle de coût sur données semi-structurées réparties

Dans le cas de sources centralisées, le médiateur a connaissance des paramètres internes, des modèles de coût et des statistiques des sources qui lui sont reliés. L'évaluation du coût d'une requête et son optimisation ne présentent dans ce cas que peu de difficultés. Dans le cas de sources hétérogènes, les sources sont autonomes et ne fournissent pas toutes les informations dont le médiateur aurait besoin lors d'une phase d'optimisation. Certaines sources peuvent être de fonctionnalités limitées si bien que même les possibilités d'interrogation se trouvent limitées. Dans ce cas, c'est le plus souvent au médiateur de pallier aux déficiences fonctionnelles.

À ces problèmes présents dans le contexte de sources hétérogènes, s'ajoutent les problèmes dus au contexte semi-structuré. L'optimisation s'avère très complexe : en effet, parmi toutes les façons possibles de résoudre une requête, comment trouver la plus optimale ? Quel modèle de coût appliquer sur des données n'ayant pas de schéma précis connu à l'avance, comment prévoir la complexité d'une requête sur une source aux fonctionnalités limitées (ex. source web), dans un contexte où la disponibilité et le temps de réponse ne sont pas toujours prédictibles ?

1.2.5 Utilisation d'un cache pour l'optimisation de requêtes

Interroger à chaque requête les sources qui peuvent être dispersées sur un réseau aussi vaste que l'Internet, peut se montrer coûteux en terme de communication et en délai d'attente. Il serait intéressant pour le médiateur, de pouvoir conserver au maximum les informations issues de requêtes précédentes afin de pouvoir les réutiliser. Dans le cas d'une nouvelle demande portant sur ces mêmes données, les accès aux bases seraient évités. Les questions relatives à une telle mémorisation sont : que faut-il mémoriser ? L'espace

de cache étant limité, quelle est la politique de rotation ? Sous quelle forme et comment stocker des données pouvant provenir de sources très différentes ? Que faire des données du cache dans le cas d'une mise à jour de la source ?

Si un cache de données semi-structurées est utilisé, de nouveaux problèmes se posent quand au stockage de ces données sur disque. Les données semi-structurées ont un schéma irrégulier qui est le plus souvent découvert en même temps que les données. Comme les métadonnées sont propres au document à insérer, il est donc indispensable de stocker une donnée avec le schéma correspondant. Or même si les données semi-structurées n'ont pas de schéma fixe, il s'avère la plupart du temps qu'une même partie de schéma (ou tout le schéma) se retrouve dans chacun des documents. Un problème soulevé est la manière de stocker des données avec leur schéma correspondant de la façon la plus compacte possible.

De plus, en vue des requêtes de navigation qui risqueraient d'être posées, il faut trouver comment structurer l'information sur disque afin de pouvoir répondre efficacement à ces demandes de navigation.

1.3 Solutions existantes

Chacune des problématiques que nous avons soulevées a été l'objet de nombreux travaux sur les données semi-structurées. Certains sont basés sur une des représentations originales des données semi-structurées dont le format est OEM. D'autres plus récents s'appuient sur le format standard XML.

1.3.1 Requête sur données semi-structurées

Afin de pouvoir effectuer des requêtes sur des données XML, de nombreux langages de requêtes ont été proposés. Certains ont été conçus plus spécifiquement pour manipuler des modèles de données de type orienté OEM (OEM-QL, Lorel [Abiteboul *et al.* 1997]), d'autres pour des modèles de données de type XML (XPath [Clark et DeRose 1999], XML-QL [Deutsch *et al.* 1998], QUILT [Robie *et al.* 2000], et tout dernièrement XQuery [W3C 2001]). De tels langages de requêtes ont fait l'objet d'une étude au sein du World-Wide-Web Consortium (W3C), et le langage XQuery a été retenu comme standard.

XQuery est un langage de requête XML fonctionnel à base d'expression de chemins, de boucles de répétition, de tests de prédicats et d'éléments de reconstruction de documents XML.

1.3.2 Médiation de données semi-structurées

La médiation permettant de fédérer plusieurs sources a surtout été étudiée dans le cadre des bases de données relationnelles (Multibase, Mermaid, InterSQL [Mullen et Elmagarmid 1993], SIMS) et objets (PEGASUS [Ahmed *et al.* 1987], IRO-DB [Gardarin *et al.* 1994], GARLIC [Carey 1995], DISCO [Tomasic *et al.* 1996]). La médiation des données semi-structurées est un travail de recherche plus récent. RUFUS [Shoens *et al.* 1993] en 1993 a été le précurseur de la génération actuelle de système de données interopérables intégrant des données semi-structurées. TSIMMIS [Chawathe *et al.* 1994] en 1994 intègre en plus le web comme source d'information. Ont ensuite suivi MIX [Bornhovd 1998], STRUDEL [Fernandez *et al.* 1998], YAT [Cluet 1998] et AGORA [Manolescu *et al.* 2001].

Un des objectifs de TSIMMIS est d'intégrer des sources qui sont très hétérogènes, qui peuvent être peu structurées et qui sont susceptibles d'évoluer rapidement. TSIMMIS a introduit le formalisme OEM comme modèle de données semi-structuré. Par l'intermédiaire de l'entrepôt de donnée LORE utilisé comme source semi-structurée du projet TSIMMIS, tout un ensemble de techniques adaptées au semi-structuré a été étudié. Notamment des modèles de coût semi-structuré, et l'implémentation des guides de données (factorisation de structure de données semi-structurées).

Le successeur de TSIMMIS, MIX utilise XML comme modèle d'échange et XMAS comme méta-langage de requête. De la sorte, tous les langages basés sur du semi-structuré peuvent être convertis en XMAS qui peut être ensuite être utilisé par le médiateur. XMAS est le langage de requête d'échange de l'architecture de médiation MIX.

STRUDEL et YAT intègrent des sources de données dans un environnement Web. Ils se basent sur une représentation de graphes afin de représenter les données semi-structurées provenant des différentes sources.

L'objectif d'AGORA est de supporter les requêtes et l'intégration de sources relationnelles et semi-structurées. Les requêtes XQuery sont transformées en requêtes relationnelles et les résultats sous formes de tuples sont ensuite retransformés en XML.

1.3.3 Algèbre d'interrogation de données semi-structurées

Plusieurs algèbres pour XML (IBM [Beech *et al.* 1999], YAT [Christophides *et al.* 2000], AT&T[Fernandez *et al.* 2001], LORE [McHugh et Widom 1999b]) ont été proposées, et seule l'algèbre proposée par l'AT&T a été retenue et publiée dans le papier de travail [Consortium 2000] en décembre 2000. D'autres algèbres comme NIAGARA [Galanis *et al.* 2001] et TAX [Jagadish *et al.* 2001] ont fait leur apparition par la suite.

L'algèbre LORE a été développée pour le système LORE avec des opérateurs phy-

siques spécialement dédiés pour ce système (utilisant les différents index et le modèle de donnée OEM). YAT transforme une structure XML en une structure tabulaire, applique les opérateurs relationnel standards et enfin reconstruit l'arbre résultat. IBM, TAX et Niagara sont des algèbres utilisant des opérateurs standards relationnel étendus au semi-structuré. AT&T se base sur une sémantique formelle de sorte à pouvoir faire correspondre XQuery et l'algèbre facilement. Les optimisations se font sur les boucles.

1.3.4 Modèle de coût sur données semi-structurées réparties

Les modèles de coût sur des données réparties ont fait l'objet de nombreuses études dans le cadre de données relationnelles et objets. Les grandes catégories qui s'en sont dégagées sont : le coût par calibration [Du *et al.* 1992], par échantillonnage [Q.Zhu et Larson 1998], par historique [Adali *et al.* 1996], adaptatif [Zhu 1995], et une généralisation du coût par calibration et par échantillonnage [Naacke *et al.* 1998] utilisée dans le projet DISCO.

Une approche sur les modèles de coût répartis sur les adaptateurs a été utilisée dans le projet GARLIC [Haas *et al.* 1997] et DISCO.

Les modèles de coût spécifiques aux données semi-structurées ont très peu été étudiés, nous pouvons néanmoins citer [McHugh et Widom 1999b] pour les opérateurs de l'entrepôt de données semi-structurées natif LORE.

1.3.5 Utilisation d'un cache pour l'optimisation de requêtes

Plusieurs types de cache ont été étudiés dans le cadre de la médiation des données. Certains se basent sur des mécanismes de réplications de données, d'autres utilisent des identifiants, et enfin d'autres ont une approche sémantique.

Les caches sémantiques exploitent la structure d'une nouvelle requête afin de déterminer si une ou plusieurs requêtes précédemment formulées ne pourraient pas répondre entièrement ou en partie à une nouvelle requête.

Plusieurs méthodes ont ensuite été proposées pour stocker les requêtes, et de nombreuses solutions ont été soumises [Bourret 2000] afin de permettre un stockage des données XML. Certaines s'appuient sur des SGBDR/SGBDO déjà existants en y rajoutant un module d'extension XML, d'autres intègrent un médiateur permettant les transitions entre des données traditionnelles et une structure XML. Enfin certaines proposent un stockage XML natif suivant des techniques variées s'appuyant le plus souvent sur un système de fichiers existant (Lore [McHugh *et al.* 1997], PDOM [Huck *et al.* 1999]) ou non (NatiX [Kanne et Moerkotte 2000]).

Le projet HERMES [Adali *et al.* 1996] introduit un cache de données sémantique pour un médiateur de données semi-structurées. Pour cela, les requêtes déjà exécutées sont stockées dans une base de données locale. Cette méthode permet, lorsque le type des requêtes demandées varient peu, de pouvoir minimiser le temps de communication pour des requêtes qui ont été entièrement ou en partie résolues lors de requêtes précédentes.

1.4 Objectif et contexte de la thèse

Un des objectifs majeur est de développer un médiateur pouvant intégrer le plus grand nombre de sources différentes possibles et s'appuyant sur XML comme modèle d'échange. Ce système devra considérer spécialement les données *semi-structurées* standardisées par le langage XML.

Nous présentons un système de médiation de sources hétérogènes à travers un système conçu initialement au laboratoire *PRiSM* en collaboration avec la société *Osis*, et repris ensuite par la société *e-XMLMedia*, ainsi qu'une architecture de gestion de données semi-structurées conçue au laboratoire *PRiSM*.

Ce travail a démarré avec le projet MIROWEB [Bouganim *et al.* 1999] [Gardarin *et al.* 1999] [Fankhauser *et al.* 1998] (projet ESPRIT-25208). Celui-ci consiste à développer une architecture de médiation entre différentes sources de données. Le modèle de données semi-structurées a été choisi comme modèle pivot. Il s'agit de permettre ensuite l'exploitation de ces données par des applications clientes dans le contexte du *web*. Cette expérience a ensuite été reprise lors de l'implémentation d'un composant médiateur par la société e-XMLMedia dans son intégration dans le projet ESPRIT XML-KM (IST-12030) Il a enfin été complété par un entrepôt natif de données XML dans le projet MUSE (projet RNTL).

Nous décrirons une algèbre basée sur des opérateurs relationnels étendus à des données semi-structurées. Ces opérateurs sont faits de sorte à prendre en compte la structure arborescente des données semi-structurées tout en réduisant les coûts de parcours d'arbres en utilisant des index appropriés.

Nous décrirons également un modèle de coût adapté plus spécifiquement à ce nouveau type de données qu'est le *semi-structurée*. Nous verrons aussi pour cela un langage d'exportation de coût basé sur XML ainsi qu'un langage d'exportation de capacité des adaptateurs.

Enfin, un troisième objectif est de voir comment un cache utilisant une base de données native semi-structurée pourrait accroître les performances du médiateur. Nous utiliserons pour cela un cache sémantique composé d'un SGBD natif XML : ReposiX. ReposiX a été conçu dans le cadre du projet RNTL MUSE au laboratoire PRiSM. ReposiX

permet de stocker du XML de façon compacte et de retrouver efficacement des documents ou des sous-parties de documents semi-structurées à partir d'identifiants uniques.

1.5 Contribution

Dans cette thèse nous définissons un cadre pour l'évaluation de requêtes hétérogènes semi-structurées. Nous définissons une architecture de médiation « tout-XML », c'est-à-dire entièrement basée sur XML. Nous montrons ainsi comment nous avons mis en place des interfaces applicatives dédiées à XML, des langages d'information de coût, de capacité, de métadonnées et de définition d'adaptateurs en XML.

Nous proposons des structures d'algèbre permettant d'évaluer simplement mais efficacement des requêtes XQuery sur des données XML. Cette structure d'algèbre est basée sur les flux de données SAX standards, et comporte un mécanisme d'indexation d'arbres rangé de façon tabulaire permettant d'allier la complexité des arbres de données semi-structurées à l'efficacité des opérations sur les tableaux.

Pour estimer le coût des requêtes, nous proposons des formules de coût pour ces nouveaux opérateurs. Afin d'intégrer les informations de coût des différentes sources, nous proposons un langage de coût adapté aux formules de coût semi-structuré. Ce langage de coût présente la nouveauté d'être lui-même exprimé en XML.

Nous définirons l'utilisation d'un cache sémantique semi-structuré pour le médiateur. Pour cela, nous montrerons comment un entrepôt natif de données semi-structurées peut s'intégrer dans une architecture de médiation en tant que SGBD local conservant les résultats de requêtes déjà effectués. Nous verrons aussi comment une nouvelle requête peut être comparée à des requêtes précédemment formulée.

Pour pouvoir évaluer les performances de notre architecture, constatant qu'il n'existe pas de bancs d'essai dédié à l'intégration de données XML, nous étendrons le banc d'essai TPC-R à ce contexte.

Enfin, nous verrons comment ont été implémentées ces solutions dans le cadre des projets MIROWEB, XML-KM et MUSE.

1.6 Organisation du document

Nous donnons ci-dessous une vue d'ensemble suivi d'un bref résumé de chaque chapitre.

Vue d'ensemble des chapitres Après la présente introduction, le deuxième chapitre dresse un état de l'art sur les données semi-structurées et l'intégration des données. Le troisième chapitre présente une architecture de médiation permettant de fédérer des sources hétérogènes structurées ou semi-structurées. Le quatrième chapitre présente une algèbre utilisée pour l'évaluation et l'optimisation de requêtes sur données semi-structurées. Dans le cinquième chapitre, nous définirons le modèle de coût associé aux données de cette architecture. Nous présenterons dans le sixième chapitre une optimisation du composant de la couche médiation par l'utilisation d'un cache sémantique. Nous présenterons ensuite dans le cinquième chapitre les prototypes sur lesquels nous avons travaillé tout au long de cette thèse et qui ont constitué les différentes étapes de notre architecture de médiation. Une évaluation de performance et une comparaison par rapport à l'état de l'art sera présentée dans le septième chapitre. Et enfin, nous conclurons dans le dernier chapitre en résumant les contributions et en proposant de nouvelles perspectives de recherche.

chapitre 2 Contrairement aux données traditionnelles, les données semi-structurées sont irrégulières : des données peuvent manquer, des concepts similaires peuvent être représentés par différents types de données, et les structures mêmes peuvent être mal connues ; cette absence de schéma prédéfini, permettant de tenir compte de toutes les données du monde extérieur, présente l'inconvénient de complexifier les algorithmes d'intégration des données de différentes sources, mais aussi les différentes opérations inhérentes à un médiateur, comme la décomposition de requêtes, l'évaluation des requêtes, l'intégration des schémas et des données et enfin la recomposition des résultats. Nous présenterons dans ce chapitre un état de l'art sur les données semi-structurées et sur les architectures de médiation.

chapitre 3 Outre les problèmes classiques de médiation de données hétérogènes (intégration de données hétérogènes, schémas dissemblables, découpage d'une requête en sous-requêtes calculables par les sources sous-jacentes, optimisation d'une requête distribuée différente d'une optimisation de requête centralisée où le médiateur connaît l'organisation de chaque source et différente de l'optimisation d'une requête locale puisqu'il s'agit de gérer des requêtes provenant de sites différents), il faut pouvoir tenir compte des spécificités des données semi-structurées. Nous présentons dans ce chapitre une architecture de médiation et ses différents composants. Nous décrirons le modèle de données utilisé, le langage de requête et les interactions entre les composants. Nous décrirons également les formats basés sur XML utilisés pour communiquer des informations de métadonnées, de coût et de capacité des sources des adaptateurs au médiateur *via* l'interface applicative XML/DBC.

chapitre 4 L'évaluation des requête doit se faire en exploitant au maximum les spécificités des données et permettre une optimisation efficace. Aussi des algèbres pour leur évaluation ont fait leur apparition. Nous décrirons l'algèbre utilisée pour le médiateur, et nous mon-

trerons comment son évaluation est rendue efficace grâce aux structures particulières manipulées par les opérateurs. Les opérateurs sont les opérateurs algébriques relationnels étendus au semi-structuré. Les structures utilisent un système d'indexation des nœuds, permettant d'allier les avantages de l'algèbre sur des données tabulaires, et la richesse de représentation des arbres.

chapitre 5 Les différentes manières d'exécuter une même requête, nommées aussi plans d'exécution doivent pouvoir être modélisées par un modèle de coût qui permettra par la suite d'évaluer pour chaque plan celui de coût minimum. Nous étudierons les différentes méthodes de modélisation de coût existants. Beaucoup s'appuient sur des modèles de données relationnelles, objet ou orienté-objet. Très peu de travaux ont été faits sur les données semi-structurées, et nous nous attacherons dans cette thèse à définir un modèle de coût sur un tel type de données. Les sources pouvant être très hétérogènes, elles peuvent avoir des capacités de traitement de données très différentes, mais aussi avoir des modèles de coût plus ou moins définis. Il s'agit donc de savoir intégrer ces différents paramètres. Nous présenterons des formules de coûts pour données semi-structurées.

chapitre 6 Les sources étant réparties, les temps d'accès à une source peuvent se révéler pénalisants s'il s'agit de répondre toujours au même type de requête. Pour cela, nous montrerons que l'utilisation d'un cache sémantique au niveau du médiateur est envisageable. Nous présenterons l'utilisation d'un SGBD natif XML comme cache de données. Nous décrirons également l'architecture du SGBD natif XML nommé ReposiX dans une architecture de médiation.

chapitre 7 Trois projets ont servi de fil directeur à cette thèse. Nous décrirons l'évolution de notre architecture de médiation au sein de ces trois projets. Le projet ESPRIT MIROWEB a pour l'objectif de construire un système d'accès sur le Web. Le projet ESPRIT XML-KM consiste en l'intégration d'applications existantes dont un entrepôt de données et un système d'information géographique. Le projet RNTL MUSE consiste en la réalisation d'un moteur de recherche sur des documents XML pouvant inclure des documents multimédia.

chapitre 8 Nous présenterons dans ce chapitre, différentes évaluations et tests permettant de valider nos travaux. Nous nous appuyons sur des cas d'utilisation pour une mesure « qualitative » de l'architecture, puis sur des bancs d'essai pour une mesure « quantitative » des performances.

chapitre 9 Nous conclurons dans ce dernier chapitre en résumant les contributions de cette thèse. Nous montrerons comment nous nous positionnons par rapport aux travaux

existants, et nous verrons comment nous envisageons la suite des travaux sur ce sujet.

Chapitre 2

Les médiateurs de données semi-structurées

2.1 Introduction

Les données du World Wide Web se caractérisent par des structures irrégulières dynamiques ou inconnues. Ces types de données sont communément appelées données *semi-structurées*. L'apparition du concept des données semi-structurées est considérée comme une révolution dans le monde des bases de données. En effet, en intégrant le monde documentaire et le monde des bases de données relationnelles et objets, les données semi-structurées ont pour objectif de permettre une meilleure représentation des entités du monde réel.

En contrepartie, de telles données sont complexes à manipuler par des traitements automatiques. Les langages de requête, et les techniques d'optimisation et de stockage utilisés pour les données classiques ne s'appliquent plus. Il a donc fallu adapter, voire inventer de nouveaux concepts et algorithmes pour manipuler des objets semi-structurés.

Il faut pouvoir représenter ces données. Pour cela des *modèles de données* propres à ce nouveau type de données ont été proposés. Il faut aussi pouvoir *échanger* ces données, aussi des *formats de document* représentant ces données ont dû être définis.

Lorsque le modèle de données a été formalisé, il s'agit d'*interroger* les données. Il faut donc définir un *langage* de requête applicable aux données semi-structurées. Ce langage doit être défini de sorte à exploiter au maximum les spécificités de ces nouveaux types de données.

Un changement de technologie ne peut se faire sans tenir compte de l'historique et des technologies déjà en place. Il est donc indispensable de permettre l'échange et la

coopération entre des données provenant de sources classiques (hiérarchiques, relationnelles, objets, annuaires LDAP) et de ces sources semi-structurées. Les techniques de *médiation* ont dû être adaptées pour prendre en compte ces nouveaux types de données.

Partant de ces bases, nous présentons un état de l'art sur l'intégration de données semi-structurées dans un contexte hétérogène.

2.2 Plan du chapitre

Nous introduirons la notion de données semi-structurées dans la section 2.3. Puis nous montrerons comment ont été modélisées de telles données dans la section 2.4. Nous verrons ensuite comment des métadonnées peuvent être définies ou extraites dans la section 2.5. Il faut un langage permettant d'interroger ce nouveau type de données, et plusieurs langages ont été proposés. Nous étudierons les caractéristiques des principaux langages portant sur les données semi-structurées dans la section 2.6. L'utilisation d'un nouveau type de données ne doit pas faire oublier les autres types de données existants, il s'agit de montrer comment intégrer ces données avec d'autres données réparties sur un réseau aussi vaste que l'Internet. Cette intégration sera décrite dans la section 2.7. Divers systèmes d'intégration de données semi-structurées commerciaux ou non existent déjà. Les plus intéressants sont analysés dans la section 2.8. Enfin, nous conclurons dans la section 2.9.

2.3 Notions de données semi-structurées

Les bases de données traditionnelles s'appuient sur la régularité des structures des objets qu'elles manipulent. En effet, l'une des caractéristiques principales des bases de données relationnelles est la définition de schémas fixes auxquels les données sont ensuite obligées de se conformer. Il en va de même pour les bases de données objets. Cela simplifie les traitements informatiques (stockages, interrogations) et permet des accès sur critères très rapides. En revanche, une telle régularité ne permet pas de reproduire la pensée de l'utilisateur ou le monde réel. Ceci car les bases de données traditionnelles (structurées) ne peuvent pas travailler sur des données dont le schéma n'est pas fixé *a priori*. Cette opposition entraîne par la suite les problèmes bien connus suivant :

- *la structure de données peut évoluer* : cela nécessite alors la modification du schéma. Cette modification peut s'avérer complexe à réaliser ;
- *les données peuvent ne pas se conformer exactement au schéma* : ceci nécessite le plus souvent de surdimensionner le nombre de colonnes et d'employer beaucoup de valeurs nulles ;
- *le même attribut peut avoir des types différents suivant les données* : cela nécessite

le plus souvent l'emploi du type le plus englobant (souvent le type « chaîne de caractères ») ; ceci a pour conséquence de réduire la finesse de la description et les possibilités d'interrogation ;
- *une instance d'attributs peut être mono-valuée ou multi-valuée* : les SGBD traditionnels traitent ces deux cas comme deux cas bien distincts, or il faudrait pouvoir gérer ces cas uniformément ;
- *les données peuvent être faiblement structurées (texte brut)* : ceci entraîne l'utilisation de blocs de données brutes (BLOB) qui ne permettent pas des techniques très fines d'interrogation.

Or, la majorité des informations du monde réel (et en particulier du *web*) n'a pas de structure régulière et statique comme cela est le cas pour les données des bases de données relationnelles ou objets. Les données du web sont *non-structurées* ou *semi-structurées*.

La manipulation des données non-structurées (texte brut, données binaires) reste limitée ou trop spécifique et globale. Les requêtes sur objets longs (BLOB ou CLOB) consistent surtout en la recherche par mots-clefs. Celles-ci sont les plus souvent basées sur des techniques d'indexation dans le cas des données textuelles. Il existe aussi d'autres méthodes de recherche spécifiques (par exemple, pour les images, suivant le format d'encodage, la reconnaissance de formes, etc.) dans les données non-structurées. Ces méthodes spécifiques peuvent permettre des interrogations très poussées (utilisation de thésaurus), mais restent propres aux données manipulée et se basent sur une analyse globale des données.

Les données semi-structurées ne sont pas contraintes par l'utilisation de structures aussi rigides que dans le cas du relationnel ou de l'objet. Elles bénéficient néanmoins d'une structure flexible restant assez cohérente pour pouvoir être manipulées.

Les caractéristiques [Abiteboul 1998] [Abiteboul 1997] des données semi-structurées sont décrites ci-dessous :

- *la structure est irrégulière* : une collection de données semi-structurées peut comporter des éléments hétérogènes de différents types pour représenter la même information. Un même attribut peut être mono-valué dans certaines instances et multi-valuées dans d'autres. Des informations supplémentaires (annotation, détails) peuvent apparaître à certains endroits. Enfin des éléments peuvent aussi manquer dans certaines instances ;
- *la structure est implicite* : même si une certaine structuration peut sembler présente (indiquée par l'intermédiaire d'étiquettes, de balises, de champs, etc.), il peut ne pas exister de description explicite. L'extraction et l'interprétation de la structure est un processus difficile puisqu'il s'agit à la fois d'analyser, d'interpréter les données et d'effectuer des correspondances logiques pour enfin en déduire la structure ;
- *la structure est partielle* : une partie des données peut être constituée d'informations non structurées (images, textes bruts) ;

- *le typage est irrégulier* : il n'y a pas de typage strict dû à l'hétérogénéité des données ;
- *le schéma est lâche* : dans les SGBD traditionnels une politique de typage strict est imposée pour protéger la consistance des données. Dans des données semi-structurées, des transgressions sont tolérées et conduisent à une altération du schéma ;
- *le schéma peut être antérieur ou postérieur* : la notion de schéma peut être antérieure ou postérieure à l'existence des données. Les systèmes de gestion de base de données traditionnelles sont basés sur l'hypothèse d'un schéma fixe défini avant toute insertion de données. Dans le cas de données semi-structurées, la notion de schéma est souvent postérieure à l'existence de données ;
- *le schéma est large* : en conséquence de l'hétérogénéité, le schéma est la plupart du temps très large pour englober toutes les informations des différentes instances des données. Cela peut poser des problèmes dans les formulations de requêtes par l'utilisateur ;
- *le schéma est parfois ignoré* : les requêtes exploratoires ou à base de recherches de mots sans indications précises sont très usitées. Dans ce type de requête, le schéma n'est pas utilisé ;
- *le schéma évolue rapidement* : les sources de données semi-structurées sont habituellement dynamiques. Leurs données et leur organisation changent fréquemment. En conséquence, leurs schémas sont souvent mis à jour ;
- *le type des éléments de donnée est éclectique* : la structure d'une information peut varier suivant le point de vue. Le type des objets peut changer lors de leurs traitement. Par exemple pour décrire un wagon fumeur/non-fumeur, on peut utiliser soit l'attribut `categorie` de type chaîne, et où les valeurs possibles seront les chaînes `"fumeur"` ou `"non fumeur"`, soit l'attribut booléen `fumeur` dont les valeurs seront `vrai` ou `faux` ;
- *la distinction entre schéma et donnée est floue* : dans les SGBD conventionnels, la différence entre schéma et donnée est très marquée. Nous avons vu qu'en semi-structurée ces différences s'estompent : les schémas se modifient, sont larges, les requêtes portent aussi bien sur les données que sur les schémas. De plus, en semi-structuré, la différence entre schéma et donnée n'a parfois logiquement que peu de sens.

2.4 Représentation des données semi-structurées

Une des façons de représenter une donnée semi-structurée est de les représenter sous la forme d'un graphe. Les éléments du schéma sont alors représentés sous forme d'étiquettes attachées aux arcs ou aux nœuds du graphe.

La figure 2.1 illustre le graphe semi-structuré de deux `personnes` (graphe (a) et graphe (b)) ayant chacun un `nom` et un `prenom`, mais où le reste des informations varie.

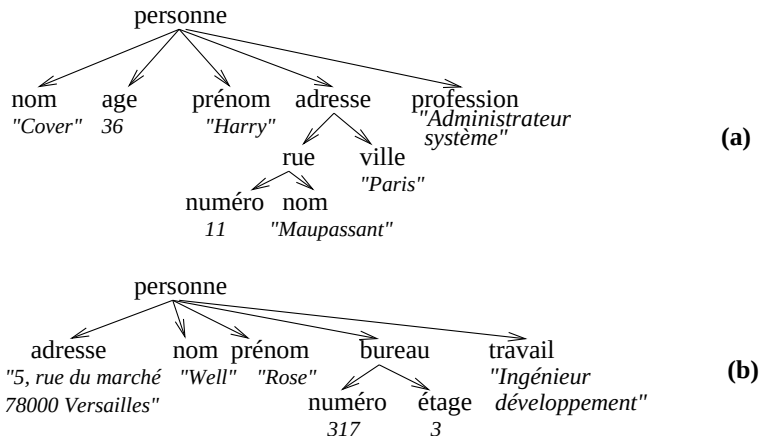


FIG. 2.1 – Exemple de graphe représentant deux données semi-structurées

Ainsi, dans la première instance (a), la `personne` est décrite par un `nom`, un `age`, une `adresse` et une `profession`. L'`adresse` est elle-même décrite par une `rue` et une `ville` et une `rue` est définie par un `numero` et un `nom`. Dans la deuxième instance (b), la `personne` est décrite par une `adresse`, un `nom`, un `prénom` un `bureau` et un `travail`, et son `bureau` est déterminé par un `numéro` et un `étage`.

2.4.1 OEM

La modélisation la plus commune des données semi-structurées est le modèle OEM présenté dans le projet d'intégration TSIMMIS [Papakonstantinou *et al.* 1995]. Dans ce projet, les données semi-structurées sont modélisées sous forme de graphe orienté. Une donnée OEM est représentée par une collection d'objets. Chaque objet peut être atomique ou complexe. La valeur d'un objet atomique répond à un type de base (entier, chaîne de caractères, image, son, etc.). La valeur d'un objet complexe est un ensemble de couples (nom d'attribut, objet). Le graphe comporte une racine.

La figure 2.2 est un exemple de donnée OEM. Les chiffres 1, 2, 3, etc. sont des identifiants d'objets (OID). La racine 1 désignée dans le schéma par l'étiquette « `divertissement` », regroupe deux objets de type complexe étiqueté « `restaurant` » et un objet simple étiqueté « `bar` » de type chaîne de caractères et de valeur « `Wild Geese` ».

La représentation textuelle de la donnée semi-structurée modélisée par le graphe de la figure 2.2 est représentée sous la forme décrite dans le document 2.1.

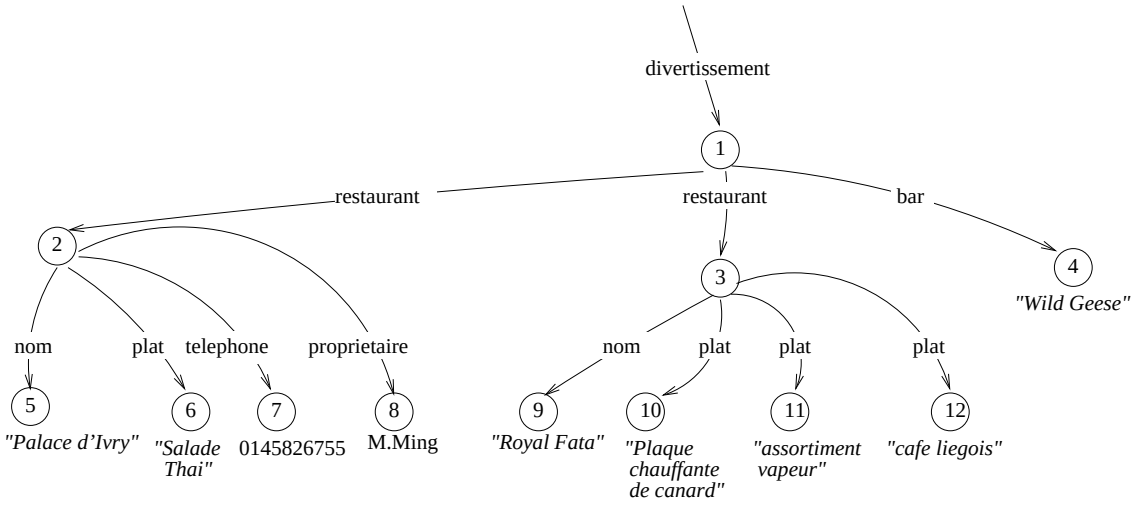


FIG. 2.2 – Exemple de graphe OEM

```
<divertissement, set, 2, 4>
    2  is <restaurant, set, 5, 6, 7, 8>
    3  is <restaurant, set, 8, 9, 10, 11, 12>
    4  is <bar, string, "Wild Geese">
    5  is <nom, string, "Palace d'Ivry">
    6  is <plat, string, "Salade Thai">
    7  is <téléphone, string, "0143669986">
    8  is <propriétaire, string, "M. Ming">
    9  is <nom, string, "Royal Fata">
    10 is <plat, string, "Plaque chauffante de canard">
    11 is <plat, string, "Assortiment vapeur">
    12 is <plat, string, "Cafe liegois">
```

Doc. 2.1: Représentation OEM du graphe de la figure 2.2

2.4.2 XML

XML [Bray *et al.* 1998] (*eXtended Markup Language*) est un format textuel extensible de description de document défini par le W3C. De la famille des langages de marquage SGML [Goldfarb 1991] (ISO 8879 :1986), il permet de s'adapter à quasiment tous les domaines où l'on a besoin de structurer de l'information de façon portable.

XML permet de faire le lien entre un langage conçu plus spécialement pour le formatage de documents (SGML) et un modèle de données en émergence permettant une vision plus réaliste mais plus complexe des données qu'est le modèle semi-structuré. Ce langage permet ainsi de définir une structure de données et son contenu.

XML est conçu de façon à faciliter l'intégration et l'échange de données entre applications. Il isole le formatage et le rendu des documents par rapport à sa structure. C'est à des langages de style spécifiques tels que XSL (*eXtended Style Sheet*) [Clark et Deach

2001] qu'on laisse le soin de s'occuper du rendu de la page XML lors de la publication.

XML est un langage à base d'éléments, d'étiquettes, d'attributs et de valeurs. Les *balises* (*tag*) ouvrantes (resp. fermantes) sont constituées d'*étiquettes* (*label*) représentées entre le symbole `<` (resp. `</`) et le symbole `>`. Le composant logique compris entre une balise ouvrante et une balise fermante est appelé *valeur*. Le composant logique constitué de la balise ouvrante, de la valeur et de la balise fermante est appelé *élément* (*element*). La valeur peut être vide, contenir du texte, d'autres éléments ou contenir un mélange des deux (*mixed element content*). Les balises définissent la structure du document. L'élément de plus haut niveau englobant tous les autres et n'ayant pas de parents est appelé *élément racine*. Un élément peut contenir des informations additionnelles appelées *attributs* (*attributes*). Un attribut est un couple formé d'un nom et d'une valeur et est représenté à l'intérieur de la balise ouvrante sous la forme `nom = "valeur"` Un document XML est un ensemble d'éléments ainsi imbriqués.

Un document XML peut avoir deux qualifications, il peut être :
- *bien formé* : quand il respecte la syntaxe du langage XML définie par le W3C ;
- *valide* : quand il est associé à une définition de type de document et qu'il la respecte (nom des éléments, type, répétition et ordre d'apparition dans le document).

Un document XML *bien formé* est un document XML qui respecte certaines règles simples :

1. Il existe un et un seul élément racine qui contient tous les autres éléments.
2. Les balises sont correctement imbriquées : chaque balise ouvrante a une balise fermante associée et il n'y a pas de chevauchement.
3. Le nom des balises est libre mais il contient au moins une lettre.
4. Les attributs des balises, lorsqu'ils existent, doivent comporter obligatoirement une valeur qui doit toujours apparaître entre double apostrophes.
5. Quand un élément est vide, les balises peuvent être simplifiées : `<balise></balise>` est identique à `<balise/>`.

La représentation XML du graphe de données semi-structurées de l'exemple de la figure 2.2 est donnée dans le document 2.2. Un attribut `categorie` (prenant la valeur '3' puis '5') a été aussi rajouté à l'élément `restaurant` (les attributs n'ont pas d'équivalents en OEM).

XML est à présent le format standard utilisé pour représenter des données semi-structurées, et [Goldman *et al.* 1999] montre que les projets utilisant OEM peuvent migrer aisément vers XML.

```
<?xml version=1.0" encoding="ISO-8859-1" standalone="yes"?>
<divertissement>
    <restaurant categorie="3">
        <nom>Palace d'Ivry</nom>
        <plat>Salade Thai</plat>
        <telephone>0143669986</telephone>
        <proprietaire>M. Ming</proprietaire>
    </restaurant>
    <restaurant categorie="5">
        <nom>Royal Fata</nom>
        <plat>Plaque chauffante de canard</plat>
        <plat>Assortiment vapeur</plat>
        <plat>Cafe liegois</plat>
    </restaurant>
    <bar>Wild Geese</bar>
</divertissement>
```

Doc. 2.2: Exemple de document XML

2.5 Métadonnées pour données semi-structurées

Les données semi-structurées sont auto-descriptives : le schéma est défini dans les données, et aucune structure n'est précisée *a priori*. Ceci permet une grande flexibilité dans le traitement (requêtes, stockage, chargement), les mises à jour et les changements structurels de telles données. En contrepartie, une telle souplesse comporte les inconvénients suivants [Suciu 1998] :

- *le stockage des données est inefficace* : le schéma doit être répliqué pour chaque donnée ;
- *les requêtes sont complexes à évaluer de façon optimale* : même une simple expression de chemin rationnelle implique souvent le parcours complet du graphe ;
- *les requêtes sont complexes à formuler* : l'utilisateur ne peut s'appuyer que sur des informations incomplètes ou inexistantes pour pouvoir formuler une requête pertinente.

Des observations sur les applications exploitant les données semi-structurées montrent que, malgré tout, les données possèdent souvent une certaine structure régulière. Il conviendrait de tirer parti de cette dernière afin de résoudre les problèmes énumérés ci-dessus. Des recherches ont été effectuées afin de décrire et d'exploiter cette régularité sans toutefois utiliser un schéma prédéfini rigide. Deux principales approches ont été proposées :

- *un schéma déduit* : calculé automatiquement *a posteriori* à partir des données. Ces schémas sont rigides et décrivent la structure de données de façon assez précise, et sont recalculés ou mis à jour régulièrement. On peut citer les *guides de données* (*dataguides*) [Goldman et Widom 1997]), les T-indexes [Milo et Suciu 1999] et les types datalog unaires [Nestorov *et al.* 1997] ;
- *un formalisme de schéma flexible* : défini *a priori*. Il permet de décrire les données en offrant un degré de précision modulable sur la structure de données. On doit à l'aide de ce formalisme, aussi bien décrire un schéma rigide et typé, qu'un schéma

n'imposant aucune structuration dans les données. De telles formulations ont fait l'objet d'étude et de standardisation, notamment DTD [Bosak *et al.* 1998] et XML-Schema [Thompson *et al.* 2001] [Biron et Malhotra 2000].

On appellera *forêt* une collection de documents XML de même nature stockés ensemble.

2.5.1 Guide de données

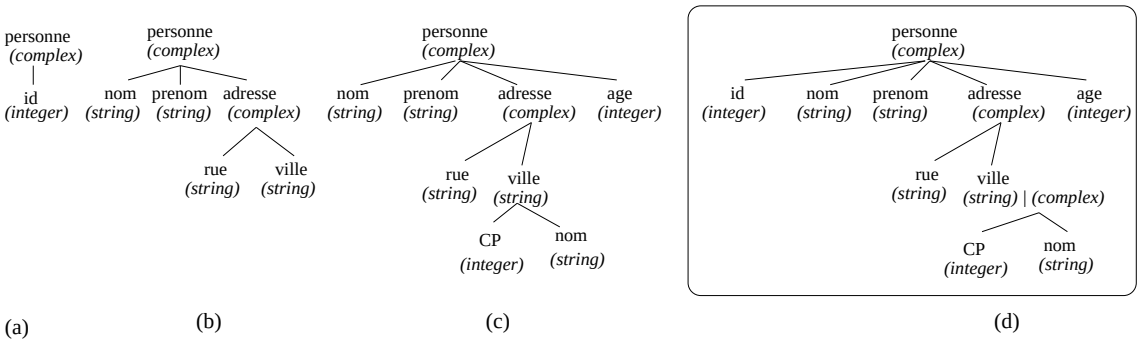


FIG. 2.3 – Schéma commun

Dans un SGBD relationnel ou objet, le schéma est considéré comme *a priori*. Les tables sont définies avant d'insérer les données. Les données semi-structurées présentées sous forme de documents XML peuvent répondre à un certain format prédéterminé, notamment afin de définir le type de certains éléments (chaîne de caractères, entier, type flottant, type complexe). La DTD et son évolution vers XML-Schema sont des formats permettant de spécifier les types. Que ce soit un schéma défini au niveau SGBD, une DTD ou un XML-Schema, même si les définitions de types peuvent être très lâches (XML-Schema permet des descriptions optionnelles ou alternatives), la structure de données garde une certaine rigidité quand à l'insertion de nouvelles données dont la structure ne s'adapte pas à celle définie pour son type. C'est pour cette raison que [Goldman et Widom 1997] a défini un *guide de données* (*dataguide*) basé sur le principe de « factorisation » des chemins communs *a posteriori*. Un guide de données est un schéma englobant tous les documents concernés. Ainsi dans la figure 2.3, les données (a), (b), et (c) ont pour guide de données (d).

Le dataguide est plus flexible, mais ne permet pas d'obtenir une définition précise des types. Ainsi, si l'on récupère la valeur 4242, comment peut-on déduire si le type est une chaîne, un nombre entier ou un flottant ? (par défaut, on prendra une chaîne), de plus des problèmes complexes de résolution de cycles peuvent survenir. La mise à jour d'une telle structure est coûteuse à maintenir.

Le guide de donnée est un schéma faible généré à partir d'un ensemble de docu-

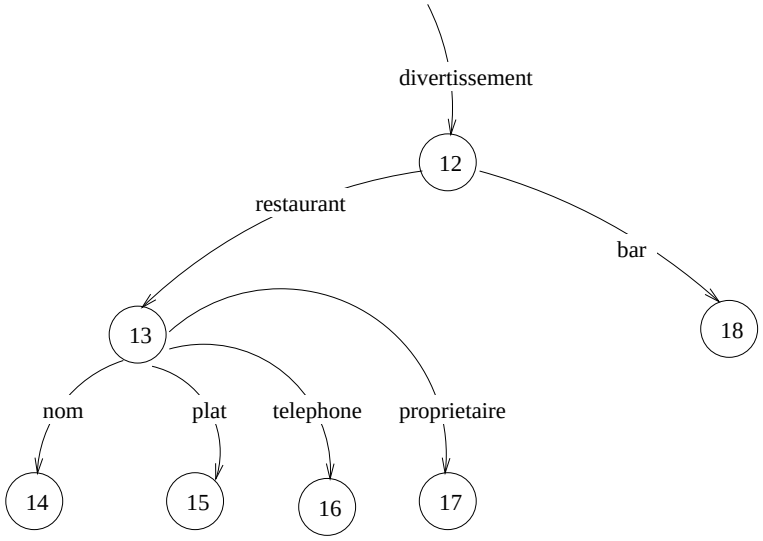


FIG. 2.4 – Guide de données du document 2.1

ments par union des arbres de structure décrivant tous les cheminements possibles dans la collection et par typage de données en texte.

2.5.2 DTD

Un DTD (*Document Type Definition*) permet aux utilisateurs de définir leurs propres balisages, attributs et entités pour des documents de types SGML ou XML.

Un document DTD est signalé par une déclaration de type de document (*Document Type Declaration*) par l'intermédiaire de la chaîne `<!DOCTYPE`. Il peut être défini explicitement ou en référence vers une entité externe. Un DTD définit une structure de document par l'intermédiaire d'un ensemble de déclarations de balisage. Une déclaration de balisage peut être une déclaration de type d'élément, une déclaration de liste d'attributs, une déclaration d'entité ou une déclaration de notation.

Déclaration de type d'élément Elle se signale par la chaîne `<!ELEMENT`. On déclare ensuite le nom de l'élément suivi des spécifications de son contenu. Si l'élément est de type chaîne de caractères, on utilisera la chaîne `#PCDATA`. Dans le cas où l'élément comporte des sous-éléments, ceux-ci sont spécifiés. Il est ensuite possible de définir le nombre d'occurrences des sous-éléments avec les symboles : '`+`', le sous-élément est représenté une fois au moins ; '`*`', le sous-élément est représenté zéro ou plusieurs fois ; '`?`', le sous-élément est optionnel ; sans symbole, le sous-élément est représenté exactement une fois.

Déclaration de liste d'attributs On introduit une déclaration d'attributs par la chaîne `<!ATTLIST` et le nom de l'élément concerné. Vient ensuite l'ensemble des attributs de cet élément, suivi chacun d'une définition de type et d'un critère existentiel (`#REQUIRED` si l'attribut est obligatoire, `#FIXED` si celui-ci est fixé, et `#IMPLIED` si l'attribut n'a pas de valeur par défaut et peut être déclaré ou non).

Déclaration d'entité La chaîne `<!ENTITY` permet de définir une constante. La valeur de l'entité peut ensuite être appelée en précédant le nom de l'entité par le symbole '`&`'

```
<!DOCTYPE bibliotheque [

<!ENTITY nom_bibliotheque "Bibliotheque Universitaire de Versailles">
<!ENTITY responsable "John Doeuf">
<!ENTITY email "John.Doeuf@bib.uvsq.fr">

<!ELEMENT bibliographie (livre *)>
<!ELEMENT livre (date+, titre, auteur*)>
<!ATTLIST livre reference CDATA #REQUIRED
                categorie CDATA #REQUIRED
                pages CDATA #IMPLIED>
<!ELEMENT date (#PCDATA)>
<!ELEMENT titre (#PCDATA)>
<!ELEMENT auteur (prenom, nom, adresse?)>
<!ATTLIST auteur pays CDATA #IMPLIED>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT adresse (#PCDATA)>

]>
```

Doc. 2.3: exemple de DTD

Dans l'exemple 2.3, le type de document de `bibliotheque`, se définit comme suit : La racine de l'arbre et l'élément `bibliographie` qui compte zéro ou plusieurs `livres`. Un livre est constitué d'au moins une `date`, d'exactement un `titre` et de zéro ou plusieurs `auteurs`. Un livre comporte également les attributs obligatoires `reference` et `categorie` ainsi que l'attribut optionnel `pages`. Les éléments `date`, `titre`, `nom` et `prénom` sont des éléments simples de type chaîne de données. Un `auteur` a un `nom` et un `prenom` obligatoire et une `adresse` optionnelle. L'élément `auteur` a aussi un attribut optionnel `pays`. Enfin trois entités `nom_bibliotheque`, `responsable`, et `email` sont définies et leurs valeurs sont respectivement `"Bibliotheque Universitaire de Versailles"`, `"John Doeuf"` et `"John.Doeuf@bib.uvsq.fr"`.

2.5.3 XML-Schema

XML-Schema [Thompson *et al.* 2001] [Biron et Malhotra 2000] est un standard permettant de définir et de typer un document XML dans un formalisme proche de celui

des bases de données. Il devient alors possible de définir des schémas de bases de données comme dans un modèle objet.

Les éléments et les attributs globaux sont créés par des déclarations qui apparaissent directement à l'intérieur de l'élément `schema`. Une fois déclaré, un élément ou un attribut global peut être référencé dans une ou plusieurs déclarations en utilisant l'attribut `ref`. Les éléments sont déclarés en utilisant le mot-clef `element` et les attributs sont déclarés en utilisant le mot-clef `attribute`. Des types simples, tels que `string`, `token`, `int`, `decimal`, `time`, `date`, `ID`, `IDREF`, etc. sont prédéfinis dans XML-Schema. De nouveaux types simples peuvent être définis par dérivation des types simples déjà existants.

Les nouveaux types complexes sont créés en utilisant l'élément `complexType` composé, dans la plupart des cas, d'une série de déclarations d'éléments et d'attributs et de références d'éléments. Ces déclarations sont rassemblées dans des groupes modèles permettant de définir, un modèle de contenu constitué d'une séquence `sequence` (ensemble ordonné), d'un choix exclusif `choice` ou d'un ensemble non-ordonné `all`.

Le nombre minimum (resp. maximum) de fois qu'un élément peut apparaître est déterminé dans sa déclaration par la valeur de l'attribut `minOccurs` (resp. `maxOccurs`). Cette valeur peut être un entier positif comme par exemple 42, ou encore le mot `unbounded` qui signifie qu'il n'y a pas de valeur limite. La valeur par défaut dans les deux cas (`minOccurs` et `maxOccurs`) est 1.

Le document 2.4 est le XML-Schema du document XML présenté dans le document 2.2.

Enfin un schéma permet de typer une requête et de vérifier sa validité avant exécution. Le typage d'une requête se fait en déduisant à l'aide du schéma, les différents domaines de valeurs utilisées. Par exemple, sur le schéma 2.4, on formule une requête cherchant les noms de restaurants inférieurs à l'entier 5 ($restaurant.nom < 5$). L'analyseur de la requête, va typer à l'aide du schéma, l'élément `restaurant` et vérifier que l'élément `nom` est bien un sous-élément possible de `restaurant`, ce qui est vrai. Ensuite, l'analyseur va analyser l'élément `nom` et voir que son domaine de valeur correspond à l'ensemble des chaînes de caractère. Or on le compare à un entier (< 5). L'analyseur va donc pouvoir déduire que la requête est incorrecte avant même d'avoir entamé son exécution.

2.6 Langage de requête pour données semi-structurées

L'interrogation des données semi-structurées nécessite la définition d'un nouveau langage de requête. Ce qui est attendu d'un tel langage de requête [Abiteboul 1997] :

- *des opérateurs standards de requêtes sur base de données* : les opérateurs habituels en base de données comme les projections, sélections, jointures, etc. doivent être

```
<?xml version='1.0'?>

<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="divertissement">
    <complexType>
      <sequence>
        <element name="restaurant" minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="nom" type="string" minOccurs="1"
                                                maxOccurs="1"/>
              <element name="plat" type="string" minOccurs="0"
                                                 maxOccurs="unbounded"/>
              <element name="telephone" type="number"/>
              <element name="proprietaire">
            </sequence>
            <xs:attribute name="categorie" type="xs:int"/>
          </complexType>
        </element>
        <element name="bar" type="string" minOccurs="0"
                                          maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

Doc. 2.4: Exemple de XML-Schema (.xsd)

définis ;

- *la possibilité de naviguer dans les données* : le langage doit intégrer des notions de chemins ;
- *la recherche par motif* : le langage doit permettre une recherche dans le style de la recherche d'information. Il doit permettre la recherche par mot-clef ou motif ;
- *la possibilité d'interrogation simultanée du schéma et des données* : il faut pouvoir combiner des interrogations sur les éléments du schéma et des interrogations sur les données dans la même requête ;
- *la construction du résultat* : il faut pouvoir spécifier la façon dont le résultat devra être présenté.

Plusieurs langages répondant plus ou moins à ces critères ont été proposés pour permettre l'interrogation de données semi-structurées. Les premiers langages ont été des extensions du langage objet OQL : SGMLQL, HyOQL et LOREL. [Christophides *et al.* 1994] a proposé en 1994 une extension de OQL avec des traversées de graphes généralisés et des recherches de contenus. En 1995, le langage fonctionnel SGMLQL [Maitre *et al.* 1997] réalise des requêtes sur des documents SGML. HyOQL [Gardarin et Yoon 1996] définit des méthodes de traversées de graphes et LOREL/OEM-QL [Abiteboul *et al.* 1998] permet des constructions de graphes généralisés sur le modèle de données OEM.

2.6.1 OEM-QL

Le langage OEM-QL (*Object Exchange Model Query Language*) a été conçu à l'université de Standford en 1996 pour interroger des graphes OEM à partir du langage OQL. C'est un langage comportant des mécanismes puissants de cœrcicion afin de gérer la manipulation de types hétérogènes.

Il permet de gérer efficacement des expressions de chemins. Pour cela, il définit la notion d'*expression de chemin simple*. Une *expression de chemin simple* est une séquence d'étiquettes séparées par des points décrivant un parcours dans le graphe OEM. OEM-QL étend ensuite cette notion avec l'introduction d'*expression de chemin généralisé* faisant intervenir certains motifs à la place de quelques étiquettes. Ces motifs ont des significations particulières comme un chemin optionnel (`?`), l'opérateur de répétition de Kleene (`*`), une chaîne de caractère quelconque (`%`), une disjonction (`|`), etc.

Exemple OEM-QL : *(Q1.) Renvoyer le nom de la personne dont la profession ou le travail contient le mot « ingénieur »*

```
select personne.nom
where personne(.profession|.travail)? like "%ingénieur%"
```

2.6.2 XPath

XPath [Clark et DeRose 1999] est un langage de requête minimal. Une expression XPath est similaire à l'expression de chemin généralisé de OEM-QL avec une syntaxe différente. Une expression XPath utilisée comme requête peut être appliquée à un document ou une collection de documents. Son résultat sera un ensemble de sous-arbres répondant aux critères de l'expression. Par exemple :

```
//livre/auteur/nom
```

renvoie tous les noms d'auteur d'un livre, et

```
//livre [@id = "04242"]
```

renvoie tous les livres dont l'attribut `id` vaut `04242`.

2.6.3 XML-QL

Pour les requêtes plus complexes, XPath ne suffit pas. Plusieurs propositions de langage d'interrogation de données semi-structurées basé sur XML ont été faites [Bonifati et Ceri 2000]. Parmi cela, on peut citer XML-QL [Deutsch *et al.* 1998] d'AT&T, XQL [Robie *et al.* 1998] de Microsoft, QUILT [Robie *et al.* 2000] et XMAS [Luascher *et al.* 1999]. Le langage XML-QL a longtemps été la proposition la plus prometteuse. La majorité des projets développés à cette époque a adopté XML-QL comme langage de requête.

Ce langage est basé sur la reconnaissance de motif dans un graphe (*pattern matching*). Le langage XML-QL est déclaratif, intègre tous les opérateurs relationnels pour l'extraction, la conversion et la transformation de données XML. Comme précédemment, il est également possible de formuler des requêtes sur des expressions de chemin. Le langage permet de plus la construction de documents, le groupement avec des requêtes imbriquées et les fonctions de Skolems.

Le langage XML-QL est similaire à l'approche « requête par l'exemple » (*query by example*) utilisée dans beaucoup de langages commerciaux. Une requête formulée dans ce langage comporte deux parties : une partie qui *filtre* le document XML par des variables, et une partie qui *construit* le résultat à l'aide des variables. Les données d'un document sont extraites en utilisant la clause WHERE en spécifiant le modèle des éléments qui doivent être utilisés et en faisant correspondre leurs contenus à des *variables*. La construction du résultat se fait en donnant le modèle du résultat final et en utilisant les variables de la partie condition ou des constantes.

Considérons le document XML « `bibliographie.xml` » se conformant au DTD donné en 2.5.

```
<!ELEMENT bibliographie (livre *)>
<!ELEMENT livre (date, titre, auteur*)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT titre (#PCDATA)>
<!ELEMENT auteur (prenom, nom, adresse)>
<!ELEMENT prenom (#PCDATA)>
<!ELEMENT nom (#PCDATA)>
<!ELEMENT adresse (#PCDATA)>
```

Doc. 2.5: DTD du document « bibliographie.xml »

Dans la requête Q2-1, les variables `$d` et `$t` sont liées respectivement aux contenus de `date` et `titre` des éléments `livre` dont le nom de l'`auteur` est « `Card` ».

Exemple XML-QL : *(Q2-1.) Renvoyer la date et le titre des livres écrits par un auteur dénommé « Card ». Le résultat devra être présenté avec des balises en anglais.*

```
WHERE
    <livre>
        <date> $d </date>
        <titre> $t </titre>
        <auteur>
            <nom> Card </nom>
        </auteur>
    </livre> IN "bibliographie.xml"
CONSTRUCT
    <book>
        <date> $d </date>
        <title> $t </title>
    </book>
```

Comme dans OEM-QL, la structure ne peut pas être complètement connue lors de la formulation de la requête. On peut de la même façon utiliser des expressions rationnelles dans les balises pour traverser des chemins arbitraires (requête Q2-2).

Exemple XML-QL : *(Q2-2.) Renvoyer la date et le titre des livres où le nom de « Card » apparait à un endroit quelconque.*

```
WHERE
    <livre>
        <date> $d </date>
        <titre> $t </titre>
        <*> Card </*>
    </livre> IN "bibliographie.xml"
CONSTRUCT
    <livre>
        <date> $d </date>
        <titre> $t </titre>
    </livre>
```

XML-QL permet d'exprimer des jointures. Celles-ci s'effectuent simplement par la réutilisation des variables dans des conditions. Les restrictions se font en utilisant la variable dans un prédicat. L'exemple Q2-3 illustre un exemple de requête faisant intervenir des jointures et des restrictions.

Exemple XML-QL : *(Q2-3.) Renvoyer les emplois des auteurs des livres dont la date est postérieure à 1993 et dont le titre contient le mot « LINUX »*

```
WHERE
    <livre>
        <date> $d </date> IN $d > 1993
        <titre> $t </titre> IN contains ($t, "LINUX")
        <auteur>
            <prenom> $p </prenom>
            <nom> $n </nom>
        </auteur>
    </livre> IN "bibliographie.xml"
    <personnel>
        <prenom> $p </prenom>
        <nom> $n </nom>
        <emploi> $e </emploi>
   </personnel> IN "personnels.xml"
CONSTRUCT
    <auteur>
        <prenom> $p </prenom>
        <nom> $n </nom>
        <emploi> $e </emploi>
    </auteur>
```

XML-QL permet les requêtes imbriquées dans la clause CONSTRUCT afin d'imposer des contraintes additionnelles aux données sélectionnées. La requête Q2-4 récupère les variables `$t` et `$d` indépendamment et le résultat contiendra des combinaisons des éléments `titre` et `date`.

(Q2-4.) Regrouper les dates de publication des livres écrits par « Card » par titre.

```
WHERE
    <livre> $l </livre> IN "bibliographie.xml",
    <titre> $t </titre>
    <auteur>
        <nom> Card </nom>
    </auteur> IN $l
CONSTRUCT
    <resultat>
        <titre> $t </titre>
        WHERE
            <date> $d </date> IN $b
        CONSTRUCT
            <date> $d </date>
    </resultat>
```

Enfin, à l'aide de fonctions de Skolem, on peut regrouper des éléments. La fonction de Skolem `livreID` de la requête Q2-5 contrôle la façon dont seront regroupés les résultats. Pour chaque élément `livre` satisfaisant le modèle de la clause WHERE, un nouvel élément est créé dans le résultat si la combinaison de valeurs pour `auteur` et `titre` est unique.

Exemple XML-QL : *(Q2-5.) Regrouper les dates de publications des livres par combinaison (auteur, titre).*

```
WHERE
    <livre>
        <date> $d </date>
        <titre> $t </titre>
        <auteur>
            <nom> $a </nom>
        </auteur>
    </livre> IN "bibliographie.xml"
CONSTRUCT
    <livre ID = livreID ($a, $t)>
        <auteur> $a </auteur>
        <titre> $t </titre>
        <date> $d </date>
    </livre>
```

En résumé, XML-QL est un langage de requête assez complet qui supporte toutes les opérations de l'algèbre relationnelle. Il est adapté aux documents XML par l'ajout de fonctionnalité d'expressions de chemins et de l'interrogation des balises.

2.6.4 XQL

Il s'agit d'une variante de XPath proposée par Microsoft et d'autres constructeurs. XQL [Robie *et al.* 1998] incorpore des fonctionnalités de requêtes dans la syntaxe de feuilles de style XSL (*Extensible Style Sheet*). XSL [Clark et Deach 2001] est un langage par motif, permettant de formater des documents XML en fournissant des fonctionnalités de sélection de nœuds satisfaisant des contraintes arbitraires. XQL définit une notation compacte pour localiser et sélectionner des données dans un document XML. Il fournit pour cela un ensemble d'opérations permettant de manipuler les données XML. Comme dans XPath, la racine est indiquée par le symbole '`/`', les fils d'un élément sont désignés par l'opérateur '`/`'. Les descendants à n'importe quel niveau d'un élément sont désignés par le symbole '`//`'. Les attributs sont signalés par le symbole '`@`'. Le caractère générique '`*`' permet de désigner un élément quelconque (ou un attribut quelconque s'il est précédé par le symbole `@`'). Enfin, le nœud résultat et la racine du sous-arbre résultat sont notés respectivement avec les symboles '`?`' et '`??`' Les critères de filtrages sont exprimés entre crochets '`[...]`'. Pour ces critères, il est possible de gérer des expressions booléennes, des prédicats de comparaison, etc.

Exemple XQL : *(Q3-1.) Retrouver tous les auteurs des livres de la bibliographie dont le nom de l'auteur contient la valeur « Card ».*

```
/bibliographie/livre/auteur??/nom [text () = "Card"]
```

ou plus généralement

Exemple XQL : *(Q3-2.) Retrouver tous les auteurs dont le nom est « Card ».*

```
//auteur??/nom [text () = "Card"]
```

Un chemin XQL renvoie toujours une collection d'éléments. Le modèle XML étant ordonné, XQL permet aussi d'interroger les éléments sur leur ordre.

Exemple XQL : *(Q3-3.) Retrouver les deux premiers auteurs de chaque livre.*

```
/bibliographie/livre/auteur [index () $lt$ 2]
```

XQL offre aussi la possibilité d'utiliser les espaces de noms (*namespace*). Des fonctions d'agrégations (minima, maxima, dénombrement) sont aussi spécifiées. XQL présente l'avantage d'étendre directement les notations des URL en conservant une syntaxe simple. Il reste toutefois fonctionnellement limité, sur la restructuration en particulier.

2.6.5 XQuery

En 2001, le langage retenu par le W3C reprend les avantages de XPath, XML-QL et XQL et est nommé XQuery [W3C 2001]. Il est issu du langage QUILT [Robie *et al.* 2000] proposé par IBM et certains auteurs de XML-QL. C'est un langage de type fonctionnel fortement typé. Dans XQuery, chaque requête est représentée par une expression.

2.6.5.1 Expression XPath

Pour adresser des éléments imbriqués dans les arbres XML, XQuery utilise XPath.

```
document ("bibliographie.xml")//livre/auteur/text ()
```

ou encore

```
collection ("bibliographie")//livre/auteur/text ()
```

On notera que le mot-clef `collection` permettant de désigner une collection de documents XML n'est pas normalisé dans XQuery 1.0 mais a été proposé à titre d'extension. Il est néanmoins présent dans les dernières spécifications de XQuery avec la même signification.

2.6.5.2 Expression FLWR

Une expression FLWR est de la forme `for ... let ... where ... return`. Les expressions FLWR sont le pendant des blocs `select ... from ... ... where ... group by` de SQL, mais elles sont beaucoup plus puissantes. La forme générale plus précise d'une requête FLWR est la suivante :

```
(1) for $ var in   foret [, $ var in foret]*
(2) let $ var :=   sous-arbre
(3) where  condition
(4) return  resultat
```

L'itération **for** permet de lier un tuple de variables à des arbres instances de forêts. Chaque instance de variable représente un arbre. Les forêts sont des collections d'arbres définies par des expressions XPath de la forme :

```
collection  (" nom")/ expression de chemin XPath
```

La clause **let** permet de collecter des arbres définis par des expressions XPath dans des variables. L'expression XPath peut être soit absolue, soit définie par rapport aux variables d'itération définies dans la clause **for**. La clause **let** est optionnelle, mais peut figurer en début de requête en place de la clause **for**.

La clause **where** permet de sélectionner les types de variables pertinents pour construire la réponse. La sélection se fait par une expression logique de prédicats élémentaires.

Enfin la clause **return** permet de construire les instances des documents résultat à partir des tuples de variables liées sélectionnées.

Exemple XQuery : *(Q4-1.) Renvoyer le nom de l'auteur et le titre des livres dont la date est postérieure à 1993 et dont le titre est « Linux Kernel 2.0 »*

```
for $p  in Collection("bibliographie")/LIVRES
where
      $p/date > 1993
    and
       contains($p/titre, "Linux Kernel 2.0")
return
   <livre>
       <auteur> $p/auteur/nom/text () </auteur>
       <titre> $p/titre/text () </titre>
   <livre>
```

2.6.5.3 Imbrication

Les expressions de requêtes peuvent être arbitrairement imbriquées, de la même façon qu'en SQL. Il est possible d'imbriquer des requêtes au niveau du **for** (pour définir des variables sur des forêts calculés), au niveau du **where** (pour calculer des valeurs de prédicats), et au niveau du **return** (pour construire des documents imbriqués).

Exemple XQuery : *(Q4-2.)Trouver les livres dont le titre est « Linux Kernel 2.0 », et qui sont écrit par un auteur dont le nom est « Card ». Formater ensuite le résultat sous la forme donnée dans la clause « return ».*

```
for $p  in Collection("personnel")/personne
for $l  in Collection("bibliographie")/livre
where
   $l/titre = "Linux Kernel 2.0"
    and
   $l/auteur/nom = $p/nom
    and
   $p/nom = "Card"
return
   <auteur>
       <id>
          <nom> $p/nom/text () </nom>
          <prenom> $p/prenom/text () </prenom>
       </id>
       <livre> $l/titre/text () </livre>
   </auteur>
```

la requête renverrait :

```
   <auteur>
       <id>
          <nom>Card</nom>
          <prenom>Rémy</prenom>
       </id>
       <livre>Linux Kernel 2.0</livre>
   </auteur>
```

Exemple XQuery : *(Q4-3.) Trouver tous les éditeurs basés en france et pour chaque éditeur, afficher son nom, son adresse et tous les livres qu'ils éditent et dont le titre comporte le mot « Linux ».*

```
for $e  in Collection("editeurs")/editeur
where
   $e/pays = "France"
return
   <editeur>
       <nom> $e/nom/text () </nom>
       <adresse> $e/adresse/text () </nom>
       <livres>
            for $l  in Collection("bibliographie")/livre
            where
                contains ($l/titre, "Linux")
                and
               $l/editeur = $e/nom
            return
               <livre> $l/titre/text () </livre>
       <livres>
   </auteur>
```

la requête renverrait :

```
   <editeur>
       <nom>Eyrolles</nom>
       <adresse>61 bd Saint Germain - 75005 Paris</adresse>
       <livres>
           <livre>Linux Kernel 2.0</livre>
           <livre>The Linux Process Manager</livre>
           <livre>Administrer un système Linux</livre>
           <livre>Redhat linux 9.0 personnal </livre>
       </livres>
   </editeur>
   <editeur>
       <nom>OReilly france</nom>
       <adresse>18 rue Séguier 75006 Paris</adresse>
       <livres>
           <livre>Le système Linux</livre>
           <livre>Pilotes de périphériques sous Linux</livre>
           <livre>Le noyau Linux</livre>
           <livre>Administration réseau sous Linux</livre>
       </livres>
   </editeur>
```

2.6.5.4 Agrégats

Des fonctions d'agrégations comme **count**, **min**, **max** sont définies en XQuery.

Exemple XQuery : *(Q4-4.)Donner le nombre de livre de la collection*

```
let $l := Collection("bibliographie")/LIVRES
return
  <nombreLivres>   count ($l)  </nombreLivres>
```

2.6.5.5 Recherche textuelles

Les recherches textuelles simples s'effectuent par le prédicat **contains** dans la clause **where**.

Exemple XQuery : *(Q4-5.)Rechercher les livres contenant le mot « Linux » dans son titre et écrit après 1993.*

```
for $l in Collection("bibliographie")/LIVRES
where
       $l/date > 1993
    and
       contains($p/titre, "Linux")
return
    <livre>
        <titre> $p/titre/text () </titre>
    <livre>
```

XQuery est un langage fonctionnel dans lequel une requête est représentée comme une expression. L'expression de la requête peut être imbriquée de la même façon qu'en SQL. Une expression XQuery respecte les capacités de XML en autorisant à la fois les spécifications de ce qui sera sélectionné et le format de sortie.

Il est très souvent possible de réécrire une expression XQuery contenant des requêtes imbriquées en clauses simples plus quelques dépendances.

Avec la prolifération de documents XML, il y a eu nécessité d'effectuer des requêtes sur XML. SQL n'est possible que sur des modèles de données relationnelles, c'est pourquoi un nouveau langage de requête avec des capacités similaires mais sur des types de données différents était requis. Après des années de travail, par le W3C, les spécifications de XQuery ont permis de répondre à ces besoins.

2.7 Intégration de données hétérogènes

Les données accessibles sur le réseau peuvent prendre différentes formes et ont chacune leurs spécificités propres.

Un système de médiation est un outil puissant permettant un accès simple aux différentes informations collectées de sources de données pouvant être très disparates. Il doit intégrer des données diverses afin de pouvoir offrir à l'utilisateur une vue centralisée et uniforme des données en masquant les caractéristiques spécifiques à leur localisation, méthode d'accès et formats.

2.7.1 Diversité des sources de données

Nous pouvons communément décrire une source de données par sa *localisation*, le *type de données* qu'elle gère, ses *possibilités d'interrogation* et le *format des résultats.*

La *localisation* d'une source de données englobe tout aussi bien le référencement du site sur lequel se situe la source (URL, adresse IP + port, annuaire LDAP), que le protocole de communication utilisé (TCP/IP, IPX, Appletalk), les moyens d'accès à la base (ODBC, JDBC) ainsi que le support (pages web, SGBD).

Le *type de données* géré par une source peut être structuré (base de données relationnelles), semi-structuré (sources XML, OEM) ou non-structuré (images, multimédia, texte libre).

Les *possibilités d'interrogation* sont aussi nombreuses, et vont de langages de requêtes évolués et standardisés (SQL, OQL) ou propriétaires (Lorel) à de simples interfaces de programmation ou encore des recherches par motifs (moteur de recherche web). Nous ferons abstraction des interfaces graphiques de requêtes utilisateurs, trop spécifiques et inadaptables dans le cadre d'une intégration.

Enfin, les *formats* des résultats complètent les disparités qui peuvent exister entre les différentes sources de données. Celles-ci peuvent être formatées suivant divers standards (XML, HTML) ou encore accessibles par des structures de programmation variées (ResultSet, OEM).

Les sources de données auxquelles nous avons accès dans un contexte interconnecté *via* le *web* se sont diversifiées selon les directions que nous avons citées, de sorte que nous pouvons à présent parler véritablement de *sources de données hétérogènes.*

2.7.2 Principes généraux de la médiation

La diversification des sources de données telles que nous venons de les décrire, a conduit tout naturellement à l'idée d'offrir à l'utilisateur un système permettant d'avoir une *vue centralisée uniforme* des données.

Les sources de données sont encapsulées par un adaptateur spécifique permettant de pallier les disparités -voire les limitations des différentes bases- et d'offrir une interface *uniforme* au composant d'évaluation ; le traitement de la requête est ainsi réparti sur le médiateur et sur les adaptateurs.

Afin que l'utilisateur ait l'illusion d'un accès à une seule et *unique* base de données virtuelle, ce système doit posséder un langage de requête unique et retourner à l'utilisateur

un résultat sous un format lui aussi unique.

2.7.3 Problèmes

L'optimisation des requêtes dans des bases de données hétérogènes conduit à de nombreux problèmes qui peuvent se classer en deux catégories : l'intégration des données hétérogènes et l'évaluation de la requête.

2.7.3.1 Intégration de données hétérogènes

L'intégration des données hétérogènes consiste dans un premier temps à spécifier les entités à intégrer et dans un deuxième temps à intégrer les données elles-mêmes. Des anomalies fréquentes dans l'intégration des données peuvent alors survenir :

- *redondance des données* : ce type d'anomalie survient lorsqu'une même donnée est copiée sur différents sites ;
- *correspondance des noms* : on en distingue deux sortes :
  - *même nom d'entité mais significations différentes* : par exemple, une même entité désignant une personne sera appelée `personne` sur un site, `p` sur un autre site, et `person` sur un autre ;
  - *nom différents mais significations identiques* : par exemple sur un site, `nom` désignera l'entité correspondant au nom de famille d'une personne, et l'entité correspondant au nom d'une société sur un autre site ;
- *conflit de divergence schématique* : ce type de conflit survient lorsqu'un même type de données est modélisé différemment. Plus le modèle de données est riche, plus le risque de conflit est possible. Par exemple, l'`adresse` d'une `personne` peut être représentée par une chaîne de caractère sur un site, et par un élément complexe comportant les attributs `numero`, `rue` et `ville` sur un autre.

L'existence de ces anomalies pose alors les problèmes suivants aux outils d'intégration :

- comment « guider » l'utilisateur dans la formulation d'une requête valide ? C'est à dire, comment lui présenter les métadonnées afin qu'il puisse formuler une requête valide ?
- comment représenter efficacement les métadonnées afin d'évaluer une requête et identifier les sources nécessaires à son exécution.

2.7.3.2 Intégration de schéma

On peut classifier [Halevy 2000] les systèmes d'intégration de données suivant la relation entre les schémas des sources locales par rapport au schéma unifié global sur le médiateur. On distingue deux types de système : un schéma global comme vue sur des schémas locaux *GAV (Global As View)* ou des vues locales comme vues du schéma global *LAV (Locale As View)* (figure 2.5).

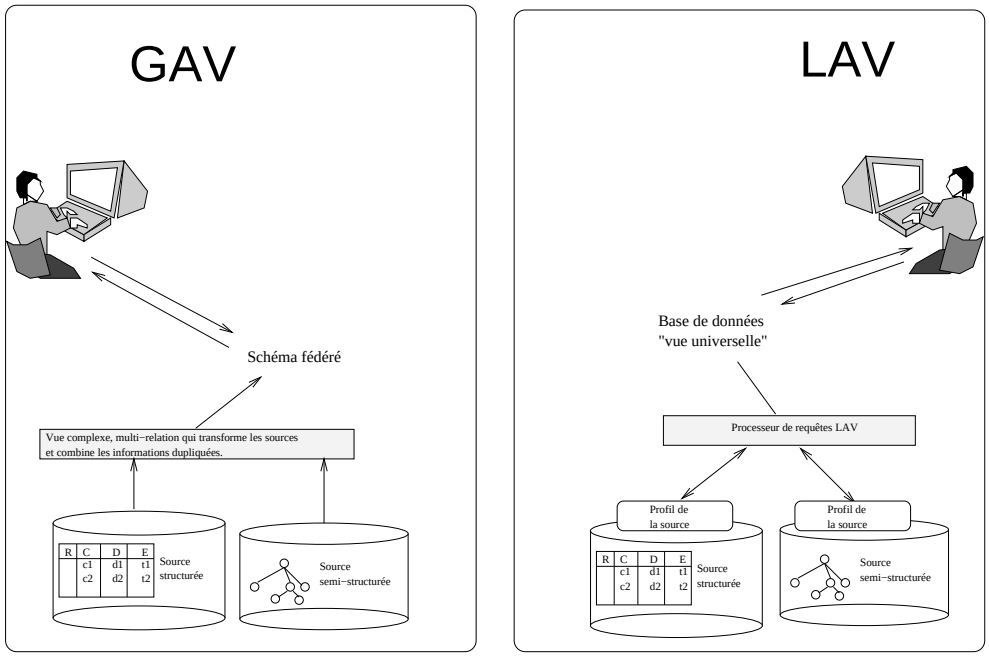


FIG. 2.5 – Comparaison d'architectures GAV et LAV

Dans l'approche GAV, la transformation d'une requête sur le schéma global en requête sur le schéma local est une simple opération faite par le gestionnaire de vue. Dans le cas d'une approche LAV, la requête sur le schéma global doit être reformulée suivant les schémas des sources locales. D'un autre côté, dans une architecture GAV, une modification sur l'ensemble des sources locales ou sur leur schéma entraîne une reconsidération complète du schéma global. Dans l'architecture LAV, chaque source est spécifiée de manière indépendante. Un changement local de schéma est considéré en mettant à jour la vue locale. De plus, si les données des sources locales n'ont pas le même format (relationnel, semi-structuré), il est difficile de définir le schéma global comme vue des sources de différents formats. En utilisant une approche LAV, chaque source peut être décrite séparément par un mécanisme de vue spécifique à son format.

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

2.7.3.3 Évaluation de requête

Le médiateur présente des vues intégrées des sources de données. Ainsi une requête formulée au médiateur est posée indépendamment de la localisation des différentes données intervenant pour calculer le résultat. Cela introduit trois difficultés :

- *la décomposition d'une requête* : il s'agit à partir d'une requête posée sur une vue intégrée, de *localiser* les données intervenant dans sa résolution, de produire des *sous-requêtes* spécifiques à chacune des sources, d'*ordonner* ces sous-requêtes et éventuellement d'introduire des opérateurs au niveau du composant de médiation afin de compléter cet ensemble de sous-requêtes. La *localisation* des sous-requêtes nécessite des structures spécifiques de gestion de méta-données ;
- *la recomposition des résultats* : une fois les sous-requêtes soumises à chacune des sources, il s'agit de savoir *recomposer* les différents résultats entre eux. Les résultats de chacune des sous-requêtes peuvent éventuellement faire l'objet d'un traitement additionnel soit parce que l'évaluateur de sous-requête n'a pas la capacité nécessaire pour traiter entièrement cette dernière, soit parce que les sous-requêtes comportent des dépendances entre elles ;
- *au niveau de l'optimisation* : le médiateur a rarement une vision sur la façon dont sont traitées les sous-requêtes au niveau des sources (placement des données, type de stockage, indexation, stratégie d'évaluation). De plus la distribution des données sur des sources disjointes ne permet pas d'utiliser directement les algorithmes employés normalement dans le cas d'un SGBD centralisé.

2.7.3.4 Cas particulier des données semi-structurées

Les données semi-structurées ont une structure de *graphe*. Ces données ayant par nature une structure qui n'est pas encore complètement connue, leurs manipulations peut s'avérer complexe dans le cas d'une *restructuration* suivant un autre schéma, ou encore dans le cas d'une *composition* de deux objets semi-structurées. Enfin lorsqu'on prend en compte dans les opérations précédentes, le cas des attributs *multivalués*. Les problèmes lors de l'évaluation de la requêtes sont alors liés à :

- *la restructuration* : comment transformer un graphe suivant une structure cible, sans perdre d'information. ;
- *la composition* : comment composer plusieurs graphes en tenant compte des branches communes. Soit la figure 2.6 (A) décrivant le schéma d'un objet de type `personne` et le schéma d'un objet de type `voiture`. Comment composer deux données répondant à ce schéma suivant le schéma représenté en (B), sachant que les attributs $Personne/Voiture/Imm$ et $Voiture/Numero$ correspondent, et les attributs $Personne/Voiture/Caract/Annee$ désigne le même objet que $Voiture/Annee$;
- *aux objets multivalués* : comment tenir compte des attributs multivalués lors de

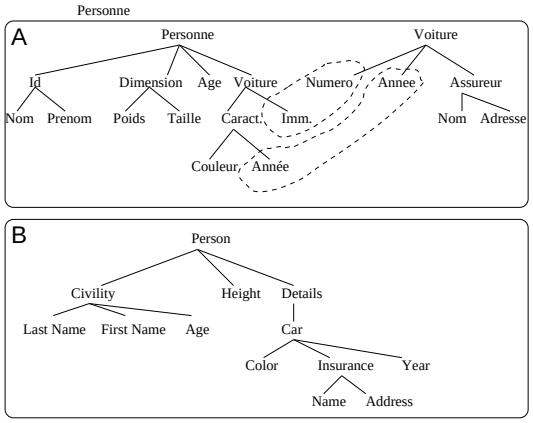


FIG. 2.6 – Schéma avant et après recomposition et restructuration

la composition, de l'évaluation ou de l'élimination des doublons.
Soit le schéma initial des données représenté en (a) dans la figure 2.7. Supposons que l'on veuille restructurer les données suivant le schéma représenté en (b) sur la même figure.

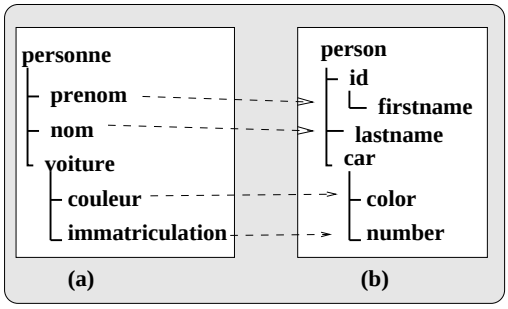


FIG. 2.7 – Schéma avant et après recomposition des données

Soit les deux données représentées en (c) et (d) de la figure 2.8 correspondant au schéma (a).
Le nœud fils `voiture` de `personne` de l'objet représenté en (d) est multi-valué. Lors de la restructuration de (d), doit-on « éclater » la structure initiale suivant en autant d'objets que d'instances de nœuds multivalués comme représenté dans la figure 2.9 (e) ou conserver le groupement comme représenté en (f) ?
Le nœud fils `prenom` de `personne` de l'objet représenté en (c) est multivalué. Un nouveau nœud `id` est placé comme parent du nœud associés à `prenom`. Dans le résultat final, doit-on dupliquer le nœud `id` (figure 2.9 (g)) ou le nœud `prenom` (figure 2.9 (h)) ?

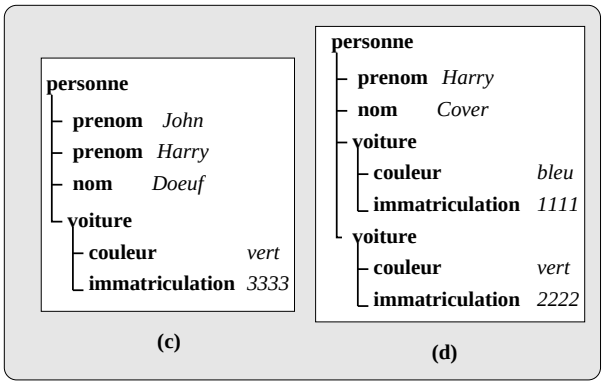


FIG. 2.8 – Données à traiter

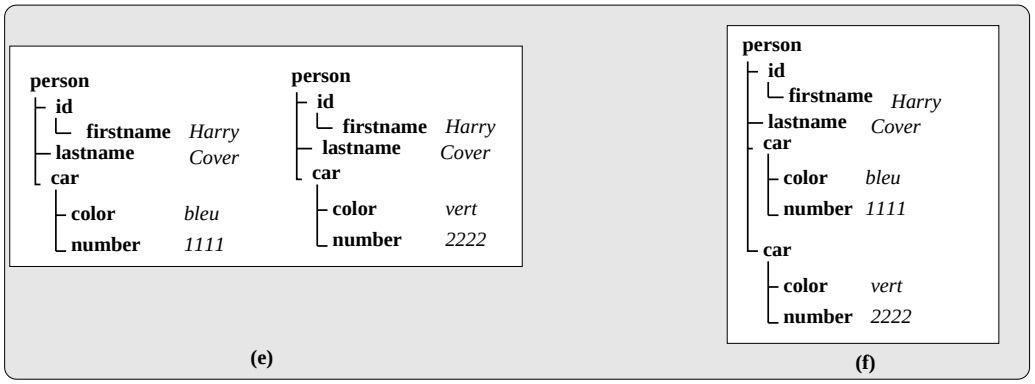


FIG. 2.9 – Restructuration de la donnée (d)

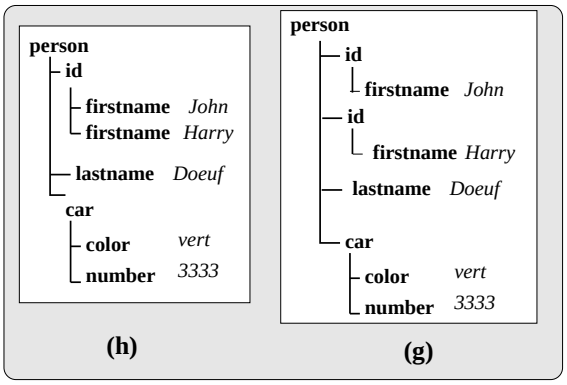


FIG. 2.10 – Restructuration de la donnée (c)

Les données semi-structurées ont une structure arborescente mal connue. Elles n'ont de ce fait aucune description précise du nombre de *niveaux* à partir d'une racine, ni de

la *largeur* maximum du graphe renvoyé. Par exemple, si on ne limite pas le nombre de niveaux ramenés, et si on choisit par exemple de récupérer la racine correspondante à la base de données semi-structurées, on peut être amené à charger la base entière. De ce fait, les coûts induits par une requête portant sur des données semi-structurées, sont extrêmement difficiles à prévoir.

Les requêtes portant sur les données semi-structurées peuvent de plus réunir des données de types différents de sorte que l'évaluation s'en trouve complexifiée. Les expressions de chemins dans les requêtes peuvent rendre encore la requête plus compliquée. Un problème majeur étant le coût d'exploration d'un arbre et l'accès à un nœud particulier suivant son chemin.

Enfin, le dernier problème induit par le concept semi-structuré est lié à la décomposition et à la recomposition d'arbres suivant un modèle, sachant que certains attributs peuvent être multivalués.

2.7.4 Architecture de médiation

Les architectures des bases de données fédérées ont progressivement convergé vers une vue unifiée proposée par DARPA et Gio Wiederhold [Wiederhold 1992] représentée par la figure 2.11.

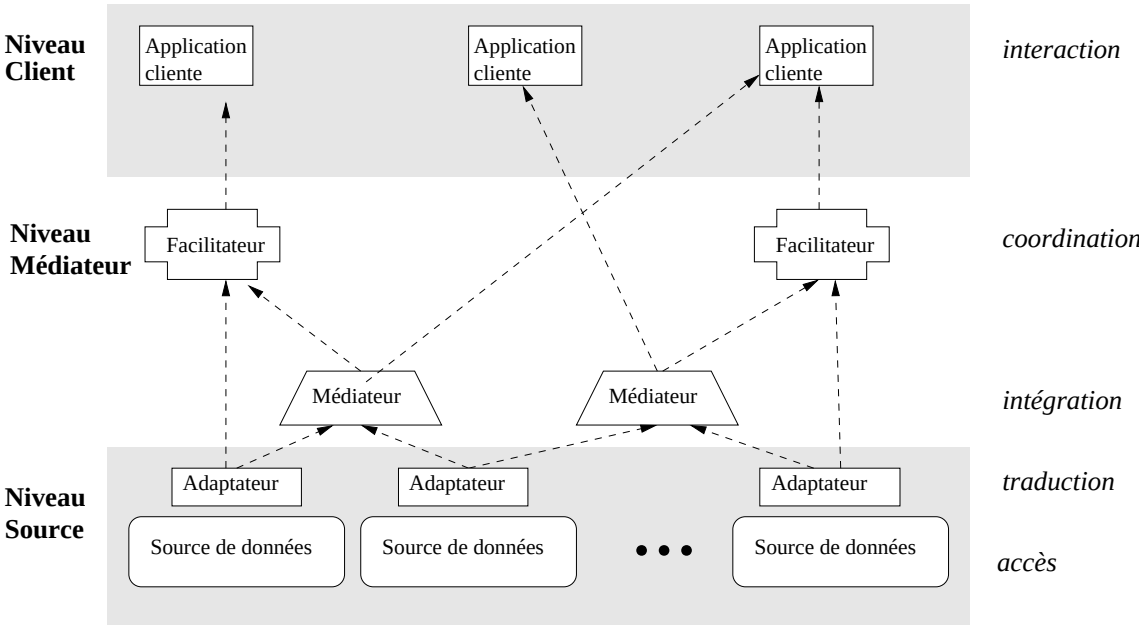


FIG. 2.11 – Architecture DARPA I3

Cette architecture est composée de trois niveaux comme suit :

1. *Le niveau source* : comporte les différentes sources de données ; à l'aide d'un *adaptateur* (*wrapper*), il est capable de communiquer avec les médiateurs et facilitateurs du niveau supérieur, en leur fournissant une vue homogène de la source à laquelle il est associé. Un adaptateur accepte une requête donnée dans le langage commun du médiateur, la transcrit dans le langage natif de la source et exécute la requête. Le résultat de la requête transmise sous forme native est alors transformé suivant le modèle de données global du médiateur et renvoyé à celui-ci.
2. *Le niveau médiateur* : comporte des *médiateurs* permettant d'intégrer les données en provenance de différentes sources afin de répondre aux requêtes des utilisateurs. Ce module joue un rôle actif dans la couche entre les applications utilisateurs et les sources de données. Son rôle est de fournir à la couche supérieure une vue centralisée des sources qui sont hétérogènes et distribuées.

   On trouve aussi à ce niveau des *facilitateurs* permettant d'identifier les sources qui peuvent être des sources de données ou des médiateurs, et de composer les réponses pour les utilisateurs.
3. *Le niveau client* : comporte les *applications* clientes (navigateurs, programmes d'application, interface graphique).

L'intégration d'une nouvelle source se résume à développer un adaptateur pour cette dernière et le rendre accessible par le médiateur [Cluet 1998].

Un langage de requête commun ainsi qu'un modèle de données commun entre les facilitateurs, le médiateur et les adaptateurs doivent être définis pour toute l'architecture. Le rôle d'un adaptateur consiste essentiellement à permettre la traduction du langage de requête commun dans le langage natif de la source qu'il gère, et la traduction du modèle de donnée de la source en modèle de données commun.

Les sources de données accessibles peuvent être très différentes : certaines sont des SGBD relationnelles, d'autres des SGBD objets, certains encore sont des systèmes de fichiers ou des pages web et d'autres des moteurs de recherche, ou encore même des médiateurs.

Ces disparités doivent être prises en compte dans le développement de l'adaptateur spécifique, mais cela n'est pas toujours évident. En effet, certaines sources (par exemple un moteur de recherche, ou une page web) offrent des capacités limitées d'interrogation (par rapport à un SGBD relationnel par exemple). Le médiateur doit pouvoir en tenir compte. Ensuite, la répartition des sources sur un réseau à l'échelle de l'Internet entraîne l'accroissement de situations d'indisponibilité des sources dues à des problèmes de communication ou de serveurs inaccessibles.

2.7.5 Synthèse

Les problèmes de l'intégration de sources de données hétérogènes se décomposent en deux catégories : l'intégration des schémas et l'intégration des données elles-mêmes. Le processus d'intégration des schémas consiste à passer de plusieurs schémas distribués et hétérogènes à un schéma homogène et intégré. L'intégration des données consiste à pouvoir définir des instances d'objets répondant au schéma homogène préalablement défini correspondant aux données des sources dont ils proviennent.

Outre ces problèmes généraux relatifs à l'intégration de données de sources hétérogènes, le cadre des données semi-structurées en pose des nouveaux. Comment intégrer des schémas à partir de schémas dont la structure est elle-même très flexible ? Comment intégrer des données pouvant être multivaluées ou ayant des structures dissemblables ?

L'intégration des données hétérogènes distribuées a donné lieu à un modèle d'architecture (DARPA I3). Ce modèle distingue deux composants principaux : les médiateurs et les adaptateurs.

Les adaptateurs sont dédiés à l'hétérogénéité des données : ils convertissent l'interface d'entrée/sortie de la source gérée en une interface commune à tous les adaptateurs et au médiateur.

Les médiateurs sont dédiés à la distribution des données : leur rôle est de regrouper les informations données par les adaptateurs.

2.8 Architectures de médiation existantes

La plupart des architectures de médiation se sont orientées vers l'architecture DARPA I3. Les premières se sont appuyées sur des sources de données relationnelles (MULTIBASE [Landers et Rosenberg 1982], MERMAID [Templeton *et al.* 1987], Information Manifold [Kirk *et al.* 1995]). Avec l'avènement des systèmes de bases de données objets ou orientées objets, il a fallu tenir compte de ce nouveau modèle de données dans les architectures de médiation (IRO-DB [Gardarin 1997], PEGASUS [Ahmed *et al.* 1987], GARLIC [Haas *et al.* 1997], DISCO [Tomasic *et al.* 1996]). Enfin, récemment, sont apparues des sources à base de données semi-structurées, et les nouvelles architectures de médiation ont dû s'adapter, aussi bien dans des projets de recherche (TSIMMIS [Chawathe *et al.* 1994], MIX [Bornhovd 1998], Y_AT [Cluet 1998] et AGORA [Manolescu *et al.* 2001]) que dans des produits industriels (NIMBLE [Nimble 2002a] [Nimble 2002b] [Nimble 2002c] et Xperanto [IBM 2002]). Nous approfondissons dans la suite, les architectures de médiation prenant en compte les données semi-structurées.

2.8.1 TSIMMIS, GARLIC et MIX

TSIMMIS (*The Stanford-IBM Manager of Multiple Information Sources*) [Chawathe *et al.* 1994] [Garcia-Molina *et al.* 1994] est un projet permettant de développer des outils qui facilitent l'intégration de sources d'informations hétérogènes comportant des données structurées et semi-structurées.

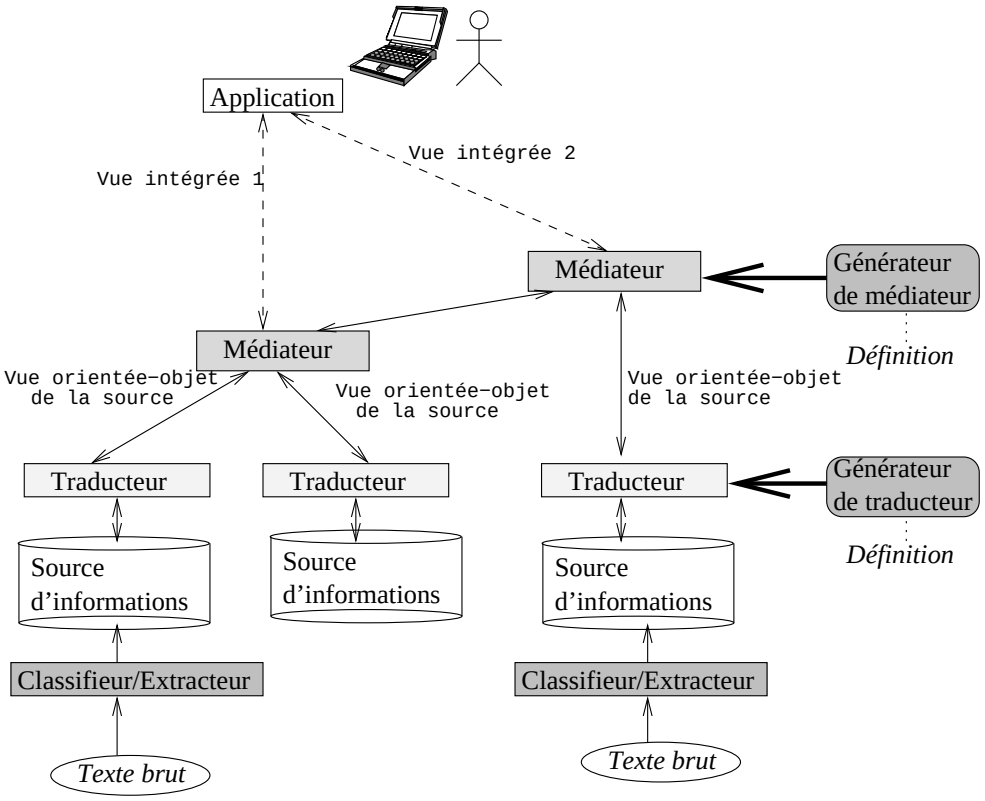


FIG. 2.12 – Architecture de TSIMMIS

De ce fait, TSIMMIS est une architecture composée des modules suivants (voir figure 2.12) :

- *classifieur/extracteur de données sur le Web* : depuis du texte brut, des pages HTML ou des pages XML. Le classifieur/extracteur identifie et classifie de tels objets (ex. est-ce un fichier texte ? une page web HTML ? un courrier électronique ?), et en extrait des propriétés (ex. date, auteur). Le classifieur/extracteur est basé sur le système RUFUS [Shoens *et al.* 1993] développé au centre de recherche d'IBM Almaden ;
- *traducteur (ou adaptateur)* : permettant la transformation des requêtes et des données. Le traducteur convertit les requêtes OEM-QL sur des données du modèle commun OEM en requêtes spécifiques à la source. Il convertit ensuite les données résultats de la source en données du modèle commun OEM ;

- *médiateur* : permettant de combiner les informations des différentes sources ;
- *applications clientes* : le médiateur tout comme les traducteurs, prend des requêtes OEM-QL en entrée et renvoie des données résultats OEM. Des applications clientes (MOBIE *MOsaic-Based Information Explorer*) permettant de naviguer dans les données à travers le web ont été développées afin d'offrir une interface conviviale à l'utilisateur final.

Un des objectifs de TSIMMIS est d'intégrer des sources qui sont très hétérogènes, qui peuvent être peu structurées et qui sont susceptibles d'évoluer rapidement. TSIMMIS a introduit le formalisme OEM [Papakonstantinou *et al.* 1995] comme modèle de données interne et a adopté un langage de requête dérivé d'OQL : OEM-QL. Afin de pouvoir utiliser une source de données purement semi-structurées, un SGBD semi-structuré natif LORE basé sur OEM a aussi été développé.

Le projet GARLIC [Haas *et al.* 1997] est un projet semblable développé au centre de recherche d'IBM Almaden. Son objectif est l'intégration de diverses sources multimédia (images, vidéo, texte, objets médicaux, cartes géographiques).

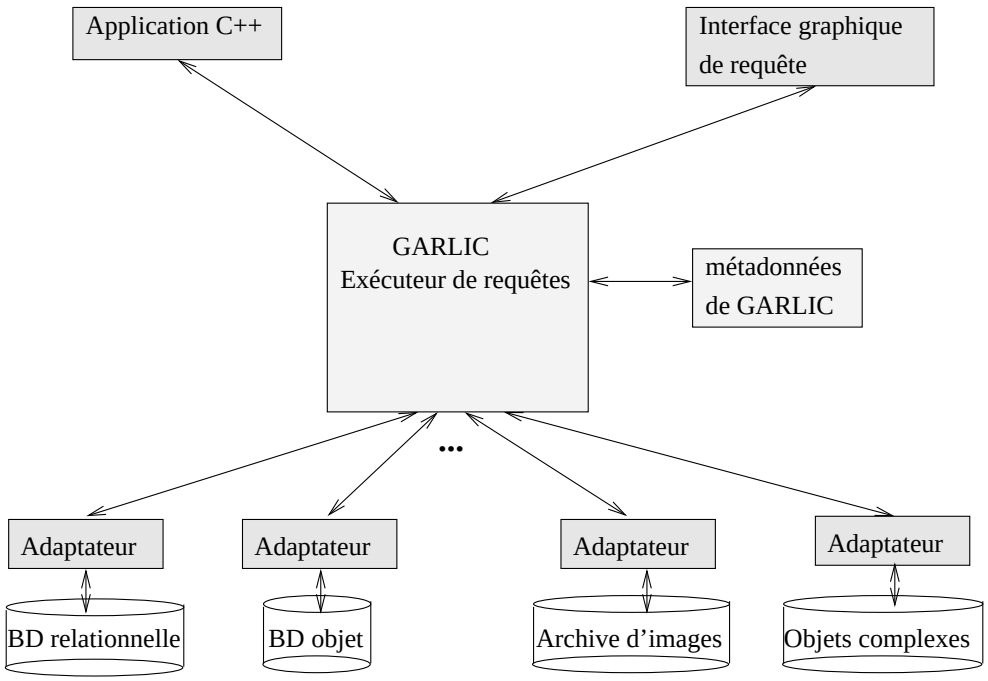


FIG. 2.13 – Architecture de GARLIC

L'architecture de GARLIC (figure 2.13), tout comme TSIMMIS, adopte l'architecture à base de médiateurs et d'adaptateurs de DARPA I3. Le modèle de données commun de GARLIC est orienté-objet, et le langage de requête sur ces données est une extension de SQL appelée GQL (*Garlic Query Language*).

Outre le modèle de données utilisé, la différence fondamentale entre l'approche de

GARLIC et celle de TSIMMIS est dans la prise en compte des capacités des sources. Le médiateur de TSIMMIS assume que les adaptateurs auxquels il adresse les requêtes décomposées savent tout faire [Li *et al.* 1998], et c'est aux adaptateurs de pallier aux défaillances des sources. Le médiateur de TSIMMIS ne prend pas en compte les capacités des sources, et donc, la décomposition et l'optimisation des requêtes est plus simple à réaliser au niveau médiateur. Il y a une bonne répartition des tâches mais l'implémentation des adaptateurs est rendu plus lourde.

Avec GARLIC, le médiateur prend en considération les capacités des sources aussi bien pour les calculs de coûts que pour les décompositions des requêtes envoyées. Son approche est plus fine et permet de traiter certaines requêtes impossibles à TSIMMIS. Les algorithmes de traitement du médiateur sont en revanche plus complexes et la centralisation entraîne un travail supplémentaire pour le médiateur. Un langage de description des capacités des sources est aussi nécessaire. Les pré-requis fonctionnels demandés aux adaptateurs de GARLIC se base sur le plus petit ensemble commun de propriétés. Au niveau des *opérations d'exécution*, il est demandé à l'adaptateur de savoir *au minimum retrouver les objets d'une collection*, et *accéder aux attributs d'un objet donné*.

GARLIC n'est pas un médiateur de données semi-structurées, mais l'intégration de sources multimédias (peu ou pas structurées) avec des sources relationnelles et objet a permis à GARLIC de détailler la définition des adaptateurs et créer un modèle d'architecture pouvant intégrer des sources très hétérogènes.

Il est à noter que TSIMMIS a ensuite adopté l'approche de GARLIC [Vassalos et Papakonstantinou 1997] et LORE a fait évoluer son format de données semi-structurées OEM vers XML [Goldman *et al.* 1999]. Cela a permis la conception de MIX [Bornhovd 1998]. MIX utilise donc XML comme modèle d'échange et XMAS (*Xml Matching And Structuring language*) comme méta-langage de requête. De la sorte, tous les langages basés sur du semi-structuré (Y_ATL, XML-QL, UnQL et MSL) peuvent être convertis en XMAS qui peut être ensuite être utilisé par le médiateur. XMAS est le langage de requête d'échange de l'architecture de médiation MIX.

2.8.2 STRUDEL

STRUDEL [Fernandez *et al.* 1998] est un constructeur de sites web permettant de définir quelles sont les données qui seront disponibles sur le site. L'idée principale est de séparer l'administration des données du site, la création et la gestion de la structure du site, et la représentation graphique des pages web finales. Pour cela, le constructeur de site crée un modèle uniforme de toutes les données disponibles sur le site. Puis, le constructeur de site utilise ce modèle pour définir la structure du site web en appliquant aux données, une requête de « définition de site ». Enfin, le constructeur de site spécifie la représentation graphique des pages en utilisant le langage de STRUDEL pour la définition de modèle (*template*) HTML. Les données résident soit sur des sources externes (SGBD,

fichiers structurés), soit dans l'entrepôt (*repository*) interne de STRUDEL. À chaque niveau du système STRUDEL, toutes les données (qu'elles soient internes ou externes) sont modélisées par un graphe orienté similaire à OEM. Un ensemble d'adaptateurs spécifiques se charge de convertir la représentation externe en graphe. La vue intégrée des données est appelée *graphe de données*.

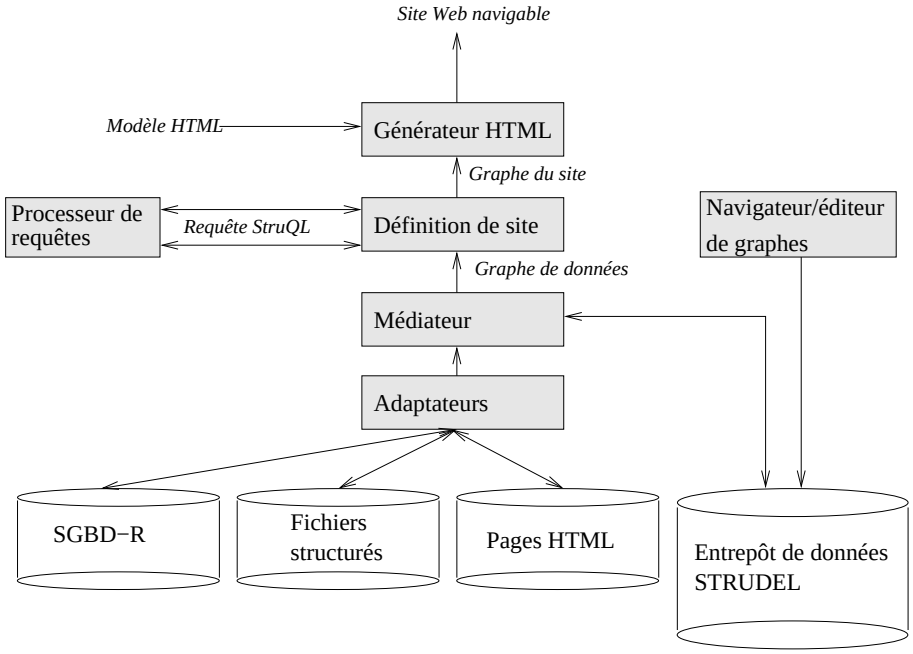


FIG. 2.14 – Architecture de STRUDEL

L'architecture de STRUDEL est représentée à la figure 2.14. Elle est composée de :

- *entrepôt de données STRUDEL* : le graphe de données d'un site web est stocké dans l'entrepôt de données STRUDEL. Les données sont obtenues des adaptateurs qui convertissent les données des sources externes en des données au format interne semi-structuré utilisé par STRUDEL ;
- *navigateur/éditeur de graphe* : il permet à l'utilisateur de créer, mettre à jour et visualiser les graphes pouvant être utilisés pour le graphe de données et le graphe du site ;
- *médiateur* : il fournit une vue uniforme des données sous-jacentes. Plutôt que d'interroger les sources externes à la demande au moment de l'exécution de la requête, son approche est d'intégrer les données en stockant préalablement les données des sources externes dans l'entrepôt de données STRUDEL ;
- *processeur de requêtes* : STRUDEL définit le langage STRUQL (*STRUdel Query Language*) pour réaliser des requêtes et restructurer des données semi-structurées. L'interpréteur de requête de STRUDEL utilise les opérateurs physiques traditionnels (jointure, restriction, sélection) ainsi que des opérateurs nécessaires pour l'interrogation de schéma (ex. trouver tous les noms d'attributs dans un graphe) ;

– *générateur HTML* : pour produire la représentation graphique de chaque page du site web, un modèle HTML est associé à chaque nœud du graphe du site. Le résultat est un site web navigable.

2.8.3 Y_AT

De la même façon que STRUDEL, Y_AT intègre des sources de données dans un environnement Web. Il se base sur une représentation de graphes afin de représenter les données semi-structurées provenant des différentes sources. Le langage Y_ATL est un langage déclaratif de conversion/intégration à base de règles ; ce n'est pas un langage de requête et il ne fournit pas de possibilités de requête comme des expressions de chemins généralisées. Cependant, il permet des primitives de restructuration et des fonctions de Skolem pour manipuler des identifiants et créer des graphes complexes.

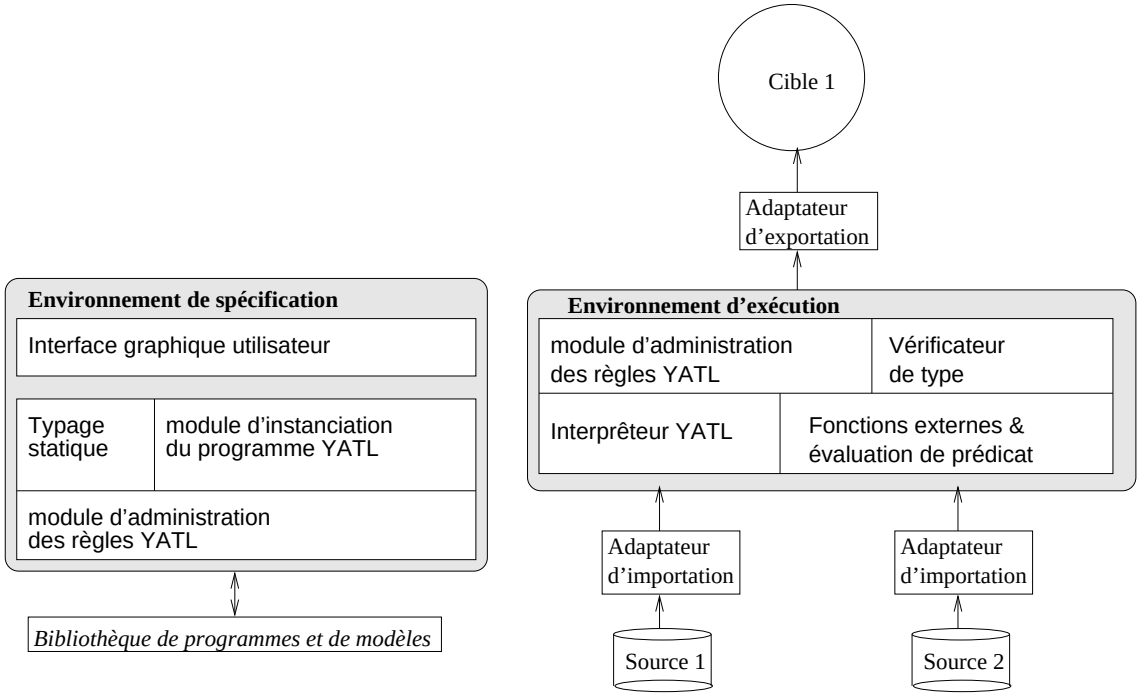


FIG. 2.15 – Architecture du système Y_AT

La figure 2.15 décrit l'architecture de Y_AT. Il y a trois parties principales : (1) l'environnement de spécification, (2) l'environnement d'exécution, et (3) une bibliothèque de programmes et format. Chacun des environnement repose sur le module de modèle Y_AT et d'administration de règles Y_ATL. L'environnement de spécification offre (i) un module permettant de vérifier ou déduire le type d'un programme, (ii) une interface graphique permettant à l'utilisateur de définir les traductions en utilisant (iii) le module d'instanciation pour personnaliser les programmes existants si besoin est.

L'environnement d'exécution utilise des adaptateurs pour importer (resp. exporter) des données depuis (resp. vers) YAT et un interpréteur pour réaliser la traduction. L'interpréteur se base sur un module séparé pour traiter des fonctions externes et des prédicats. Si cela est requis, il vérifie les types à l'aide du vérificateur de type.

2.8.4 AGORA / LeSelect

AGORA [Manolescu *et al.* 2001] est le seul projet avec Information Manifold [Levy *et al.* 1996] à utiliser une approche LAV. À la différence de ce dernier qui utilise uniquement des données relationnelles, AGORA manipule aussi des données semi-structurées. Pour cela, il s'appuie sur le médiateur LeSelect [Caravel 1998]. LeSelect est un médiateur dont le modèle de donnée est de type modèle relationnel étendu et dont le langage d'interrogation commun est de type SQL. La motivation de AGORA [Florescu *et al.* 2000] est de compléter les fonctionnalités de LeSelect en :

- définissant un schéma virtuel, générique et *relationnel* décrivant le contenu de documents XML ;
- traduisant des requêtes spécifiques à XML dans le schéma d'intégration relationnel ;
- étendant l'optimisation de requêtes afin de pouvoir gérer les cheminements ;
- utilisant l'indexation textuelle pour améliorer les performances des recherches plein texte.

L'architecture d'AGORA est représentée à la figure 2.16.

L'objectif d'AGORA est de supporter les requêtes et l'intégration de sources relationnelles et semi-structurées. Le schéma global de AGORA est une DTD XML. Les sources relationnelles et XML sont décrits comme vues sur ce schéma global. Un schéma relationnel générique est utilisée comme interface entre ce schéma et les sources. Pour chaque type de nœud XML (élément, attributs, textes, commentaire), une table relationnelle est associée dans ce schéma.

De cette façon, les requêtes XQuery sont transformées en requêtes relationnelles et les résultats sous formes de tuples sont ensuite retransformés en XML.

2.8.5 Logiciels commerciaux

Outre les projets de recherche que l'on a cités ci-dessus, de nombreuses entreprises ont conçu des systèmes de médiation de données semi-structurées. Suite à leur conception en milieu industriel, il est très difficile de trouver de la documentation quand à la conception et description des algorithmes utilisés dans ces logiciels. Nous ne détaillerons donc pas ces systèmes. On peut néanmoins citer les cas de XPeranto, NIMBLE et LiquidData qui bien que commerciaux, fournissent des spécifications techniques assez détaillées.

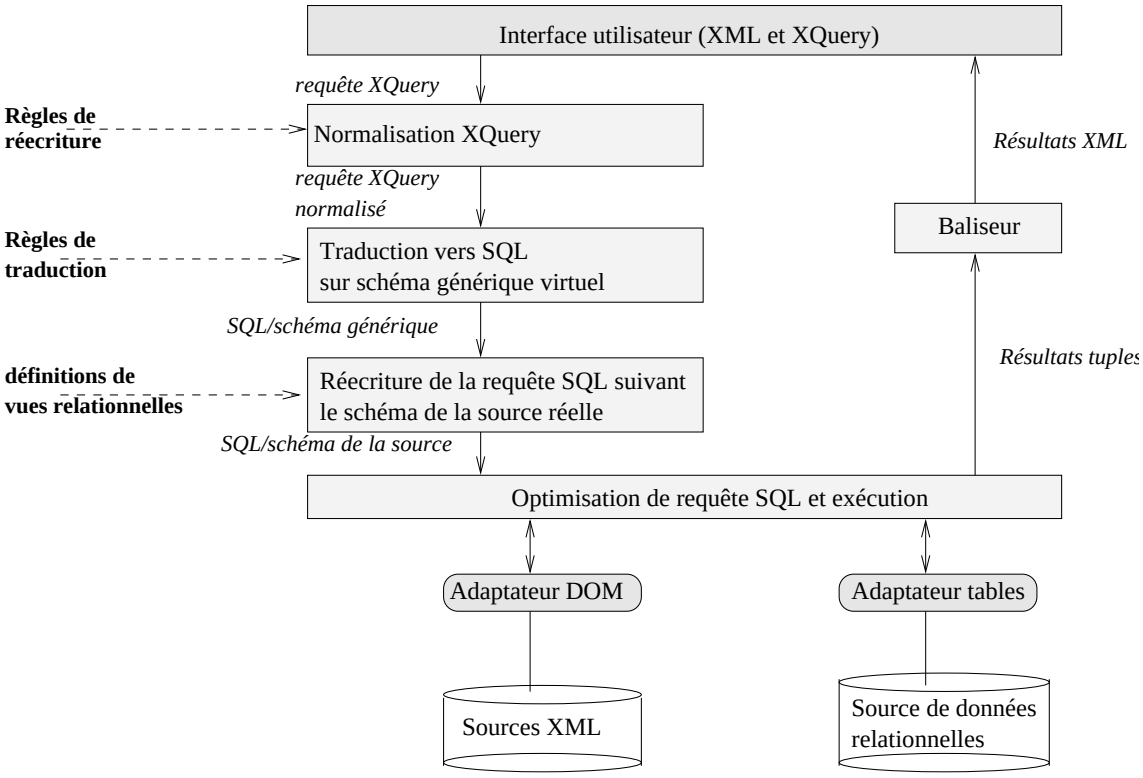


FIG. 2.16 – Architecture du système AGORA

2.8.5.1 Xperanto

XPeranto [IBM 2002] [Carey *et al.* 2000a] (*Xml Publishing of Entities, Relationships, ANd Typed Object*) est le nom de code de la future suite d'intégration de données d'IBM. Xperanto est composé d'un processeur de requêtes XQuery, d'un système de médiation permettant de récupérer des données hétérogènes de diverses sources de données (SGBD, documents de multiples formats), et enfin d'un composant à base de feuilles de style permettant de présenter les résultats sous une forme facilement intégrable dans des applications clientes. Comme AGORA, XPeranto [Carey *et al.* 2000b] transforme un langage de requête orienté XML : XML-QL en SQL. Pour cela, il passe par un langage intermédiaire XQGM (*XML Query Graph Model*) - un langage neutre de représentation intermédiaire de requête sur XML - Un composant *XML Tagger* permet ensuite de convertir la structure tabulaire des résultats SQL en document XML structuré.

La figure 2.17 décrit l'architecture d'XPeranto. L'architecture est composée de :

– *un traducteur de requête* : qui traduit la requête XML-QL en langage SQL natif pour le SGBD O-R sous-jacent. L'analyseur XML-QL prend une requête XML-QL et génère une représentation XQGM (*XML Query Graph Model*). Le composant de réécriture de requête prend la représentation XQGM de la requête, résout

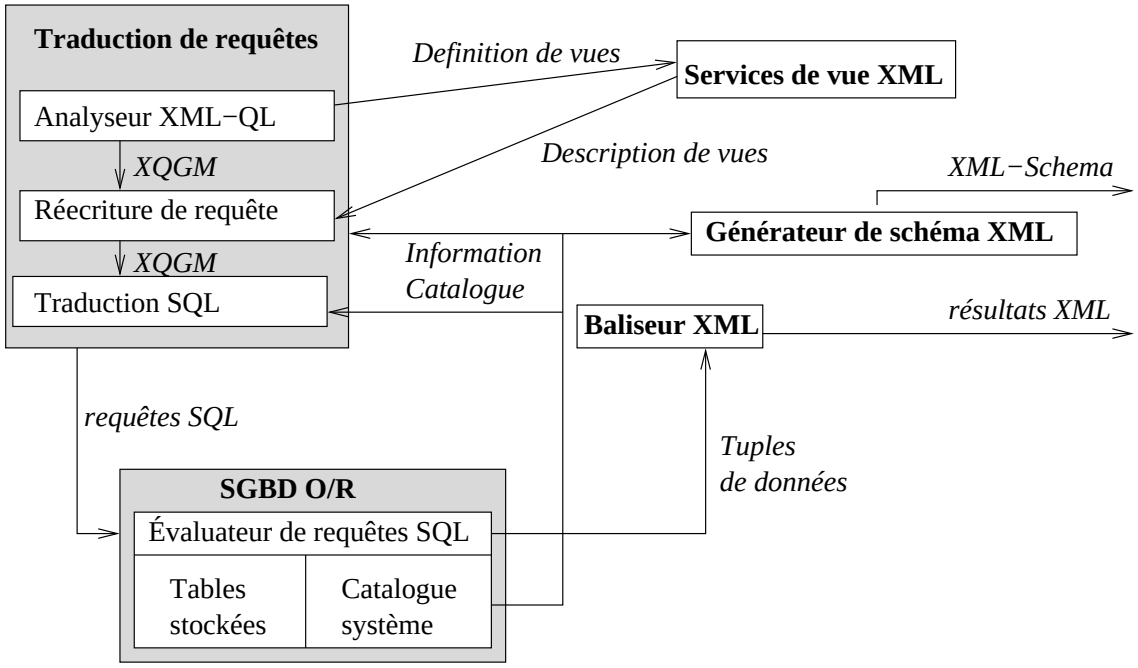


FIG. 2.17 – Architecture de XPeranto

les références sur les vues, compose la vue XML et construit une représentation sémantique équivalente XQGM. Le composant de traduction SQL traduit enfin le XQGM en instructions SQL ;

- *un service de vues XML* : fournissant une interface de stockage et de chargement de données pour les définitions de vues XML-QL ;
- *un générateur de schéma XML* : prenant les informations du catalogue des bases et produisant des informations pour les vues et résultats de requête XML ;
- *un baliseur XML* : convertissant un résultat tabulaire SQL en un document XML structuré.

2.8.5.2 NIMBLE

La suite NIMBLE [Draper *et al.* 2001] [D.Draper *et al.* 2001] est une plate-forme d'intégration basée sur XML. NIMBLE utilise XML comme format commun pour représenter des données hétérogènes distribuées. La constatation faite par Nimble est que tous les modèles réalisés pour XML sont représentés sous forme de graphe ou d'arbre et que tous sont orientés pour manipuler du XML pur. L'idée est de proposer un modèle de donnée qui s'accommoderait d'XML, mais qui traiterait efficacement les types de données les plus utilisés (relationnels, hiérarchique). Le modèle de donnée de NIMBLE est adapté aux aspects semi-structurés de XML mais est légèrement plus structuré que le modèle décrit pour XML afin de gérer plus naturellement les données relationnelles et hiérarchiques. Le

langage de requête utilisé est XML-QL au moment de l'écriture de cette thèse, XQuery étant en cours d'implémentation dans NIMBLE.

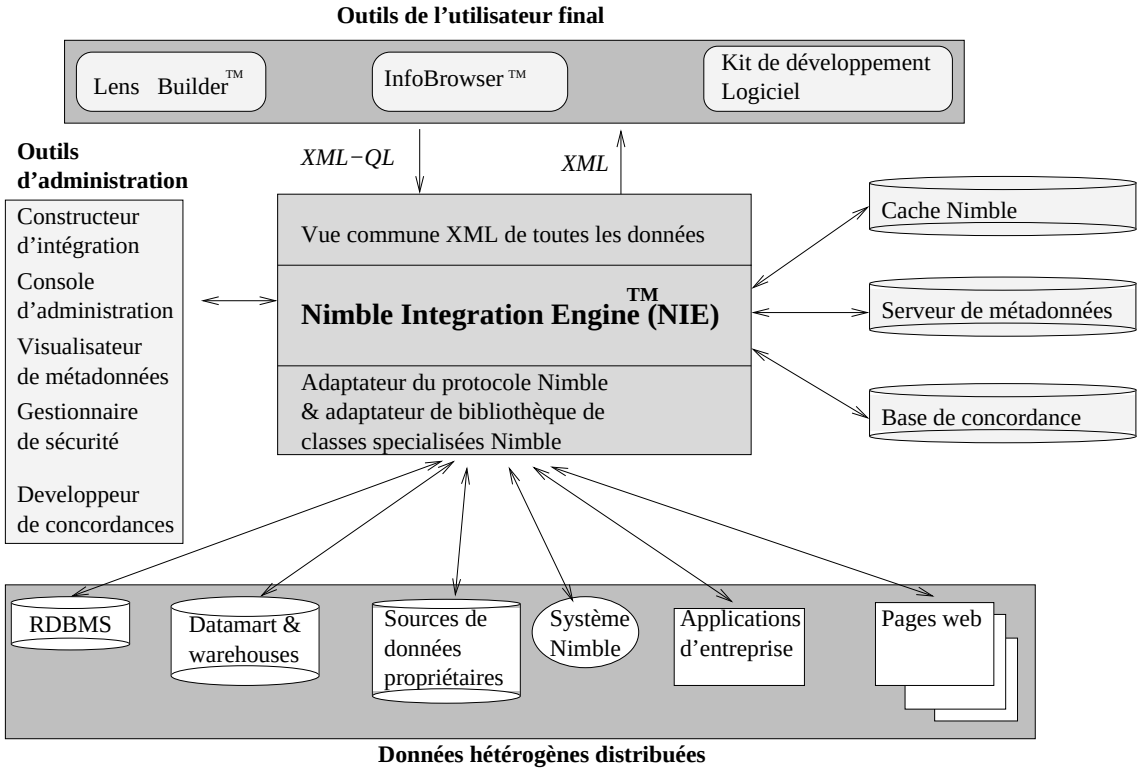


FIG. 2.18 – Architecture de NIMBLE

La figure 2.18 représente l'architecture de NIMBLE. L'architecture NIMBLE est composée de :

- *moteur d'intégration Nimble (Nimble Integration Engine)* : il comporte un compilateur de langages de requêtes, une gestion de connexion aux clients et aux stockages physiques. Le moteur d'intégration Nimble analyse et compile les requêtes en utilisant le serveur de métadonnées afin de déterminer quelle source de données est requise et si le cache Nimble doit être utilisé ;
- *cache Nimble* : contrôlé par l'administrateur. Il permet de précalculer des vues XML sur une source et de stocker les documents XML résultants dans une base de données. Une requête future portant sur cette vue XML sera ainsi redirigée vers la version précalculée dans la base de données ;
- *base de concordance* : permet de réconcilier les différences entre les diverses sources de données pour la représentation d'une même entité (ex. information sur la même personne représentée de deux façons différentes dans deux sources). Par un contrôle manuel de l'administrateur et par quelques règles prédéfinies, la base de concordance réconcilie ces différences en créant un index de clefs appropriés pointant vers les données originales ;
- *outils d'administration* : pour gérer la base de concordance, le cache, le médiateur ;

– *outils de l'utilisateur final* : pour formuler la requête de façon conviviale, développer des applications interagissant avec le moteur d'intégration de Nimble, etc.

2.8.5.3 Liquid Data

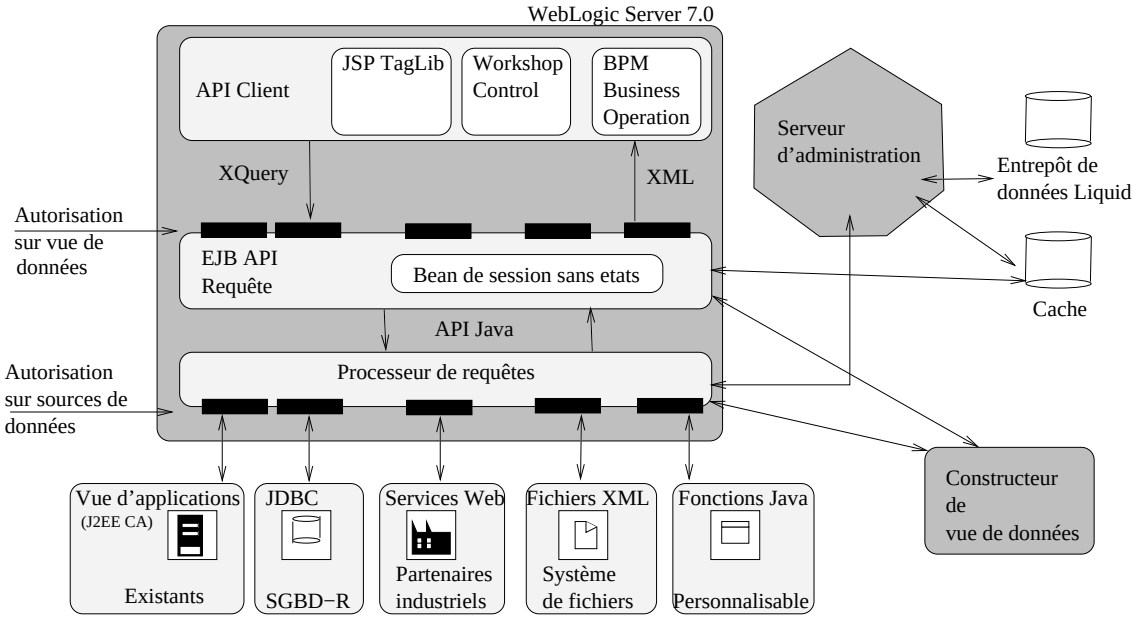


FIG. 2.19 – Architecture de LiquidData

La figure 2.19 décrit l'architecture de LiquidData [BEA 2002]. Elle adopte l'architecture DARPA I3 avec un ensemble d'adaptateurs au dessus de sources de données hétérogènes, un modèle de données commun XML et un langage de requête commun XQuery. Les applications clientes accèdent au médiateur de LiquidData en envoyant des requêtes XQuery et en récupérant des résultats XML. Le serveur LiquidData transforme la requête XQuery en un plan algébrique optimisé, qu'il transforme ensuite en un plan physique. Ce plan physique est ensuite exécuté sur les différentes sources *via* les adaptateurs, et les résultats sont remontées.

2.8.6 Synthèse

Les médiateurs prenant en compte les données semi-structurées adoptent tous l'architecture DARPA I3 à base d'adaptateurs. Tous ont représenté les données semi-structurées par une structure interne de graphe orienté étiqueté. Les précurseurs (TSIMMIS) ont défini une structure de graphe appelée OEM puis, avec l'avènement de XML, ont progressivement migré la structure de données vers le modèle de données XML. La plupart des autres

médiateurs à base de données semi-structurées se sont directement appuyé sur le format XML et ont adopté un modèle de données interne arborescent (YAT, NIMBLE). AGORA et NIMBLE ont choisi une approche relationnelle en utilisant un type de données interne relationnel et en basant les efforts sur la traduction d'une requête portant sur des données semi-structurées vers une requête relationnelle.

Les langages de requête des médiateurs ont évolué au fil de la standardisation du langage de requête. Le premier langage de données semi-structurées a été LOREL (un langage SQL étendu) utilisé par TSIMMIS, puis XML-QL, QUILT et enfin XQuery (AGORA). Des langages spécifiques comme YATL ont aussi été proposés (pour YAT).

Afin de pouvoir intégrer toute les données provenant des sources, une structure de méta-données est nécessaire. On peut différencier un schéma global comme vue sur des schémas locaux (GAV) ou des vues locales comme vues du schéma global (LAV). Seul AGORA utilise cette dernière approche dans la médiation de données semi-structurées.

Les sources interrogées par le médiateur sont diverses et peuvent être relationnelles, objets ou semi-structurées. Les adaptateurs permettant au médiateur de communiquer avec ces sources sont plus ou moins évolués. GARLIC définit un langage de communication permettant à l'adaptateur de communiquer au médiateur les opérations qu'il est capable de réaliser, ce dont le médiateur doit tenir compte.

2.9 Conclusion

Nous avons présenté dans ce chapitre une étude sur les systèmes d'intégration de base de données hétérogènes s'appuyant sur des données semi-structurées.

Contrairement aux données traditionnelles, les données semi-structurées sont irrégulières : des données peuvent manquer, des concepts similaires peuvent être représentés par différents types de données, et les structures mêmes peuvent être mal connues ; cette absence de schéma prédéfini, permettant de tenir compte de toutes les données du monde extérieur, présente l'inconvénient de complexifier les algorithmes d'intégration des données de différentes sources, mais aussi les différentes opérations inhérentes à un médiateur, comme la composition de données, l'évaluation des requêtes, etc.

Outre les problèmes classique de médiation de données hétérogènes (intégration de données hétérogènes, schémas dissemblables, découpage d'une requête en sous-requêtes calculables par les sources sous-jacentes, optimisation d'une requête distribuée différente d'une optimisation de requête centralisée où le médiateur connaît l'organisation de chaque source et différente de l'optimisation d'une requête locale puisqu'il s'agit de gérer des requêtes provenant de sites différents), il faut pouvoir tenir compte des spécificités des données semi-structurées.

Chapitre 3

Une architecture de médiation XML

3.1 Introduction

L'Internet a permis une augmentation considérable du nombre et du type de sources d'information disponibles à l'utilisateur. Par conséquent un *médiateur* fournissant un accès intégré à de multiples sources d'information devient essentiel pour la plupart des outils et utilisateurs. Par exemple un utilisateur cherchant à réserver un voyage a besoin d'un système d'intégration qui ira chercher dans diverses sources (site d'agence de voyage, site de chaîne hôtelière, horaire d'avions, logiciel d'agenda personnel) et produira un résultat intégré. L'intégration des sources de l'Internet pose de nouveaux défis. Nous distinguons les problèmes suivants plus ou moins relatifs aux sources de l'Internet par opposition à des sources fédérées que l'on retrouve dans des réseaux locaux :

- les *données semi-structurées* : les données ne possèdent pas de structures fixes (relationnelles, objets). Les traitements classiques d'évaluation et d'optimisation ne s'appliquent plus.
- l'*hétérogénéité des données* : les sources peuvent être aussi bien semi-structurées que relationnelle, objet ou textuelles. Comment gérer la diversité des schémas données par les différentes sources ? Comment intégrer les résultats de différentes sources ?
- la *communication avec des sources hétérogènes* : les sources sont accessibles de diverses manières. Certaines font appel à des interfaces de requêtes évoluées (ODBC, JDBC, IIOP) avec des langages de requêtes adaptés (SQL, OQL, XQuery) d'autre au contraire ne disposent que de moteurs de recherche ou d'interface de programmation très spécifiques.

À l'instar de médiateurs existants (TSIMMIS [Chawathe *et al.* 1994], GARLIC [Haas *et al.* 1997], STRUDEL [Fernandez *et al.* 1998], YAT[Christophides *et al.* 2000], LeSelect [Florescu *et al.* 2000], Xperanto [IBM 2002], NIMBLE [Nimble 2002a], Liquid Data [BEA

2002]), nous adoptons l'architecture à base de médiateurs et d'adaptateurs (DARPA I3 [Wiederhold 1992]).

Un médiateur est un composant qui traite de la *distribution* des données. Il interagit avec les différentes sources par l'intermédiaire d'un langage commun. L'intégration des différents résultats se fait ensuite par un traitement local au niveau du médiateur.

Pour permettre au médiateur d'interagir facilement avec les différents types de sources, l'interaction est faite par un module logiciel nommé adaptateur. L'adaptateur traite de *l'hétérogénéité* des sources. Il « traduit » le langage commun utilisé par le médiateur en langage spécifique à la source à laquelle il est connecté. De la même façon, les résultats rendus par la source sont traduit dans un modèle de donnée commun utilisé par le médiateur. Outre le passage des requêtes et des résultats, l'interaction entre le médiateur et les adaptateurs peut aussi servir à communiquer des informations relatives aux sources de données. Ces informations peuvent être :

- *les descriptions des métadonnées* : c'est-à-dire, les métadonnées ou les schémas gérés par la source ;
- *des descriptions de capacités des sources* : c'est-à-dire les types de requête que la source est capable de traiter ;
- *des informations de coûts* : c'est-à-dire les statistiques ou les formules estimant le temps de calcul de la source pour traiter un type de requête donné.

L'évaluation d'une requête au sein du médiateur est un traitement complexe. La requête doit être analysée. Les sources intervenant dans le traitement de la requête doivent être identifiées. La requête doit être décomposée en différentes sous-requêtes exécutables chacune par les sources idoines. Une phase d'optimisation est appliquée pour améliorer le temps de traitement. Enfin, après une phase de recomposition des résultats, le résultat final peut être envoyé à l'utilisateur.

3.2 Plan du chapitre

Ce chapitre est organisé de la façon suivante. La section 3.3 présente notre architecture de médiation de manière générale. Dans les sections suivantes, nous détaillons ensuite chacun des composants la constituant. La section 3.4 présente ainsi le modèle de donnée utilisé dans l'architecture de médiation. Nous expliquerons pourquoi nous avons choisi d'utiliser XML comme modèle de données. La section 3.5 expose la façon dont interagissent les médiateurs et les adaptateurs. Nous présenterons pour cela l'interface générale de communication XML/DBC, puis nous verrons comment sont définies les sources, comment elles sont interrogées, et comment les schémas et les données sont échangés. Nous verrons dans la section 3.6 comment est généré un plan d'exécution. Dans la section 3.7 nous montrerons le déroulement de traitement d'une requête. Et pour finir, la section 3.8

fait un résumé de notre architecture.

3.3 Architecture de système fédéré

La plupart des systèmes de gestion de données hétérogènes et distribuées (TSIMMIS [Chawathe *et al.* 1994], IRO-DB [Fankhauser *et al.* 1998], DISCO [Tomasic *et al.* 1996]) adoptent l'architecture à base de médiateurs et d'adaptateurs telle que définie par [Wiederhold 1992]. Dans une telle architecture les médiateurs offrent une interface uniforme permettant d'interroger des vues intégrées de différentes sources. Les adaptateurs fournissent une vue locale des sources de données suivant un modèle uniforme. L'interrogation des vues locales peut être limitée suivant les capacités relatives à chaque source.

La spécificité de notre architecture de médiation est celle d'être entièrement conçue pour gérer des données semi-structurées et plus particulièrement du XML. Nous verrons par la suite que chacun des composants classiques d'une architecture de médiation a été modifié et adapté à une solution « tout-XML ».

La figure 3.1 décrit l'architecture médiateur/adaptateurs de notre prototype.

Au niveau le plus bas se situent les *sources de données* qui stockent et manipulent les données. Au dessus de chaque source est connecté un *adaptateur*. Le rôle d'un adaptateur est de masquer les détails de l'interface de la source de données pour ne permettre que l'accès à la source de données en utilisant le protocole interne XML/DBC du médiateur. Les adaptateurs décrivent les données stockées dans la source en utilisant une description de métadonnées sous forme XML.

Le cœur de l'architecture est le médiateur lui-même. Ce médiateur est décomposé en plusieurs modules :

- l'*analyseur* : il décompose la requête de l'utilisateur en une structure interne susceptible d'être manipulée facilement par les différents composants du médiateur. Il vérifie aussi si la requête est *valide*, aussi bien *syntaxiquement* que par rapport aux types des données interrogées ;
- le *décomposeur* : son rôle est d'analyser la structure de la requête afin de déterminer comment décomposer la requête initiale en sous-requêtes ;
- le *cache des métadonnées* : le cache de métadonnées se charge de conserver au fur et à mesure les localisations des données et les schémas des différentes données réparties dans les sources ;
- le *gestionnaire de plans d'exécution* : il permet de générer l'ensemble des plans d'exécution possibles pour satisfaire une requête donnée ;
- le *gestionnaire de coûts* : il estime le coût d'exécution d'un plan ;
- l'*optimiseur* : il détermine en fonction des composants ci-dessus, quel est le plan

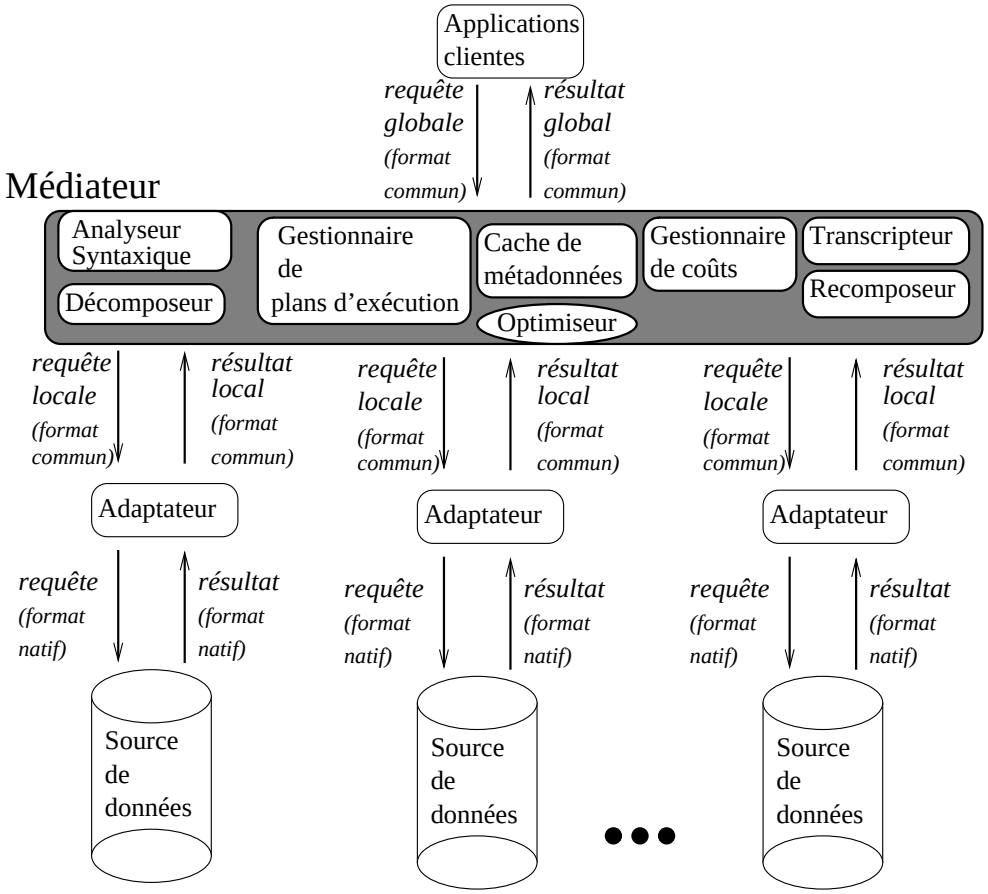


FIG. 3.1 – Architecture générale du médiateur

optimal à choisir pour évaluer la requête ;
- le *recomposeur* : il restructure et recompose les résultats donnés par les différents adaptateurs ;
- le *transcripteur* : il transforme la structure interne résultat en un format lisible par l'utilisateur (XML).

L'architecture est récursive grace à l'utilisation de la même interface au niveau de l'adaptateur et du médiateur. Ainsi le médiateur se place au cœur d'une architecture de médiation et peut être considéré lui-même comme une source de données et être interrogé par un autre médiateur *via* un adaptateur adéquat (figure 3.2).

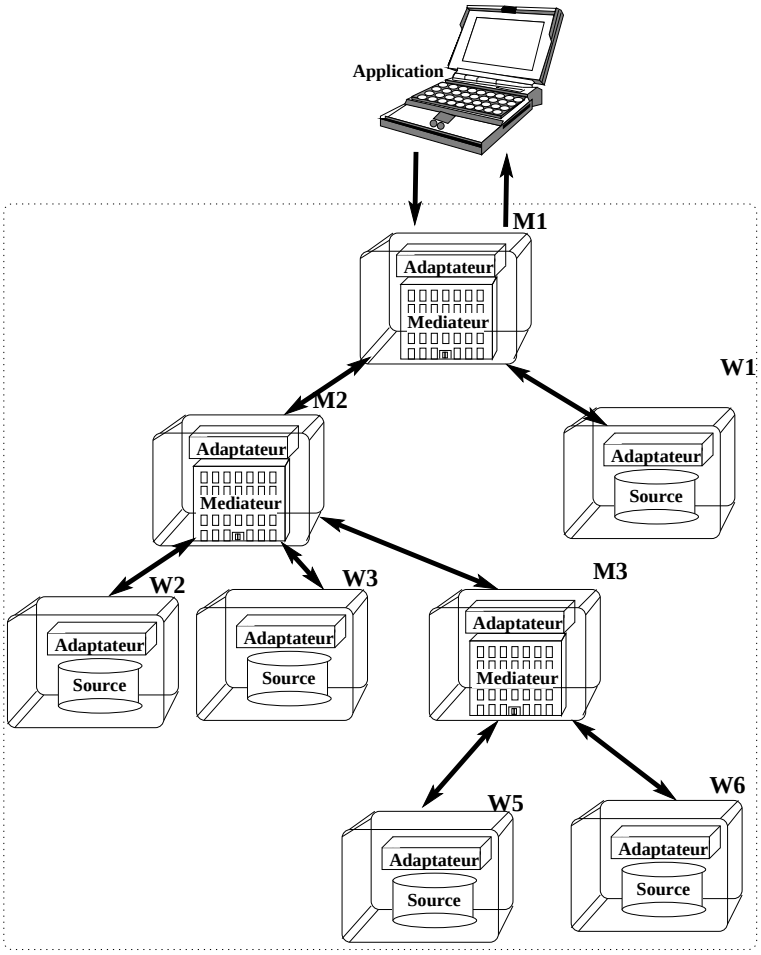


FIG. 3.2 – Interconnexion de sources/médiateurs

3.4 Modèle de données

La plupart des études sur les systèmes de gestion de données distribuées hétérogènes utilise le modèle relationnel ou le modèle objet comme modèle de données d'intégration. Notre approche est d'utiliser le standard XML[Bray *et al.* 1998] comme modèle de données d'intégration.

Les avantages de l'utilisation de XML comme modèle d'intégration tiennent à la richesse de ce langage : abondance des descriptions et typages des données, clarté, et extensibilité. Un autre critère participant au choix de XML comme modèle d'intégration est que son caractère général rend aisée la traduction des modèles de données existants. Pour chaque objet du modèle réseau, hiérarchique, relationnel, ou objet, il est possible de construire l'arbre XML associé. Les nombreux standards associés à ce langage en font le candidat idéal comme modèle d'intégration dans une architecture d'accès à des données

distribuées hétérogènes. Ainsi XML-Schema[Thompson *et al.* 2001] [Biron et Malhotra 2000] permet de fournir un modèle de métadonnées uniformes entre les adaptateurs et le médiateur, le langage XQuery[W3C 2001] offre l'interrogation efficace de requêtes sur XML et les feuilles de style XSL permettent de présenter les données à l'utilisateur.

3.5 Traitement des sources

Les différentes sources sont accessibles *via* des adaptateurs. Ces adaptateurs communiquent par une interface de programmation (API) commune appelée XML/DBC. Cette interface est inspirée de JDBC [Inc. 1997] et à l'instar de celle-ci, fournit une interface de programmation permettant d'accéder à des SGBD distants. La différence majeure résidant entre XML/DBC et JDBC est l'utilisation de requêtes XQuery plutôt que SQL et la récupération de documents XML comme résultats plutôt qu'un ensemble de tuples. Comme pour JDBC, un pilote (*driver*) pour chaque adaptateur doit être utilisé. L'accès à un adaptateur se présente sous la forme d'un couple pilote/chaîne de connexion.

La figure 3.3 illustre une application accédant à des sources de données *via* un ensemble de pilotes XML/DBC. Un pilote XML/DBC est associé à chaque type de source de données. Actuellement, il existe un pilote pour les SGBD-R (*e-XMLlizer*), pour un entrepôt de données semi-structurées (*repository v2*), et pour le médiateur. Cet ensemble de pilotes est enregistré dans un composant appelé *gestionnaire de pilote XML/DBC* qui se charge d'utiliser le bon pilote suivant la source à accéder. Cette interface uniforme XML/DBC permet à l'application d'adresser indifféremment à chacune des sources les ordres suivants :

- *ouverture de connexion* : une connexion est ouverte vers les sources concernées suivant les protocoles associés. Les authentifications nécessaires sont éventuellement effectuées ;
- *demande d'informations sur les sources* : lors de leur initialisation, les sources peuvent communiquer différentes informations au médiateur, celles-ci sont les suivantes :
  - *information sur les métadonnées* : permet à la source de communiquer toutes les informations sur la structure et les types des données qu'il gère (schéma, tables) ;
  - *information sur les capacités des sources* : chaque source n'est capable de traiter que certains types de requêtes. Ces informations de capacité doivent être communiquées au médiateur ;
  - *information sur les formules et statistiques de coût* : les sources doivent pouvoir communiquer des informations permettant au médiateur d'estimer les temps d'évaluation ;
- *exécution de requêtes XQuery* : permet au médiateur d'envoyer une requête à faire exécuter par la source ;
- *récupération du résultat en SAX ou en DOM* : récupère le résultat de l'exécution ;

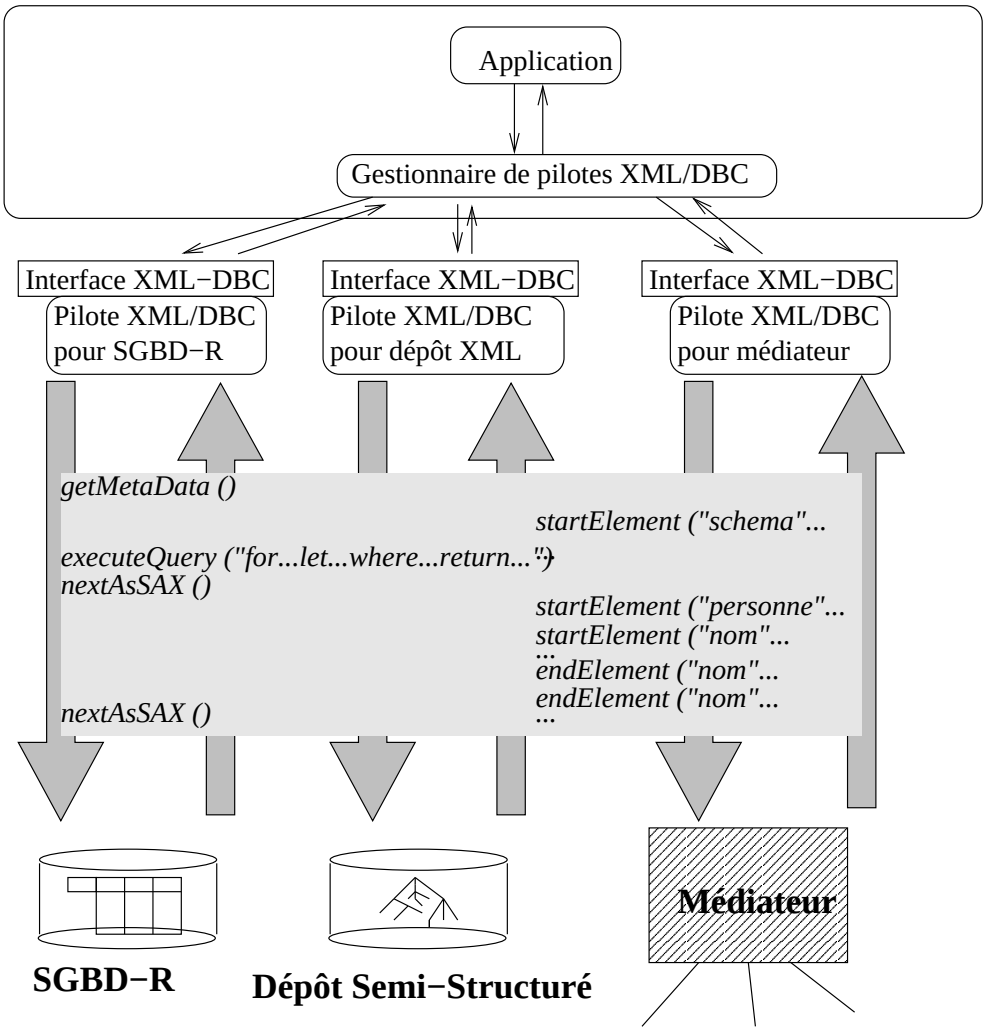


FIG. 3.3 – Accès par XML/DBC

– *fermeture de connexion* : déconnecte la source du médiateur.

3.5.1 Définition des sources de données

Les différentes sources de données doivent être accédées *via* un adaptateur dédié. L'ensemble des adaptateurs associés à un médiateur est spécifié lors de l'initialisation à l'aide d'un fichier de configuration au format XML.

Chaque méthode d'accès à un adaptateur est représentée par l'élément `subaccessor`. Elle est identifiée par un nom (valeur donnée par l'attribut `name` de l'élément `subaccessor`). Elle est accessible par l'intermédiaire d'un pilote (*driver*) XML/DBC spécifique à la source

accédée (indiqué par l'élément `driver`). L'adaptateur est localisé sur le réseau par une URI (*Uniform Resource Locator*) indiqué par l'élément `connection`. Les adaptateurs auxquels accède un médiateur peuvent utiliser une connexion spécifique de type XML/DBC :

```
xdbc:exml:extractor@darkvador.e-xmlmedia.int:WrapperSQL1
```

mais aussi des techniques d'invocation de méthodes à distance (RMI, SOAP) :

```
rmi://darkvador.e-xmlmedia.int/WrapperSQL1
```

ou encore des accès à des *servlets* (applications tournant sur un serveur web) :

```
http://darkvador.e-xmlmedia.int/servlet/WrapperSQL1
```

Le fichier XML de configuration du médiateur M2 de la figure 3.2 serait représentée par le document XML suivant :

```
<accessor type="mediator" name="M2">
  <subaccessors>

    <subaccessor name="W2">
      <driver>com.exmlmedia.extractor.ExtractorDriver</driver>
      <connection>
         xdbc:exml:extractor@darkvador.e-xmlmedia.int:W1.xml
      </connection>
    </subaccessor>

    <subaccessor name="W3">
      <driver>com.exmlmedia.repository.RepositoryDriver</driver>
      <connection>
          xdbc:exml:extractor@darkvador.e-xmlmedia.int
      </connection>
          <login>USER1</login>
          <passwd>PASSWDUSER1</passwd>
    </subaccessor>

    <subaccessor name="M3">
      <driver>com.exmlmedia.mediator.MediatorDriver</driver>
      <connection>
          xdbc:exml:mediator@godzilla.e-xmlmedia.int:M2.xml
      </connection>
    </subaccessor>
  </subaccessors>

</accessor>
```

3.5.2 Exportation d'informations

Lors de l'initialisation, l'adaptateur communique des informations au médiateur. Ces informations sont utilisées par le médiateur pour définir les décisions à prendre

lors de l'évaluation des requêtes. Ces informations sont relatives aux schémas gérés par les différentes sources, les capacités des sources, c'est-à-dire le type des requêtes qu'ils sont capables de traiter, et enfin les statistiques et les formules de coût d'évaluation des requêtes sur ces adaptateurs. Elles sont demandées aux adaptateurs par l'intermédiaire de méthodes de l'API API XML/DBC qui sont respectivement `getMetaData ()`, `getCapacity ()` et `getCost ()`.

3.5.2.1 Exportation de métadonnées

Lorsqu'un nouvel adaptateur est enregistré auprès du médiateur, ce dernier lui demande le schéma de la source *via* une méthode de l'API XML/DBC `getMetaData ()`.

L'adaptateur envoie alors le schéma demandé sous forme d'un document XML encapsulant un ou plusieurs documents XML-Schema. Ce document XML est appelé *description de métadonnées*. Si la source ne comporte pas de schéma, un schéma par défaut est alors généré par l'adaptateur. Ce schéma par défaut est calculé suivant le principe du *guide de données* (*dataguide*). Pour cela, on réalise une union des arbres de structure décrivant tous les cheminements possibles dans la collection. Faute d'information sur les types des éléments terminaux, le type générique `ANY_TYPE` est alors utilisé.

Nous allons décrire chaque partie de la description de métadonnées de l'adaptateur W3.

Une description de métadonnées est représentée par un document XML répondant à un schéma précis. À ce titre, son en-tête s'écrit de la façon suivante :

```
<?xml version='1.0'?>
<xm:metadata
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xm="http://www.e-xmlmedia.com/XMLDBC/Metadata"
  source="SourceW3">
```

L'ensemble des schémas gérés par le médiateur sont ensuite introduits dans l'élément `schemas` :

```
  <schemas>
```

L'adaptateur W3 comporte le schéma d'une collection `livre`.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.bibliographie.org"
    xmlns="http://www.bibliographie.org">
  <xs:element name="livre">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="titre" type="xs:string"/>
        <xs:element name="numero" type="xs:decimal"/>
        <xs:element name="date" type="xs:dateTime"/>
        <xs:element name="auteur">
          <xs:sequence>
            <xs:complexType>
              <xs:element name="nom" type="xs:string"/>
              <xs:element name="prenom" type="xs:string"/>
            </xs:complexType>
          </xs:sequence>
        </xs:element>
    </xs:sequence>
  </xs:element>
</xs:schema>
```

L'adaptateur W3 comporte également le schéma d'une collection `personnes`.

```
<xs:schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.qui-es-tu.org"
    xmlns="http://www.qui-es-tu.org">
  <xs:element name="personne">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="nom" type="xs:string"/>
        <xs:element name="prenom" type="xs:string"/>
        <xs:element name="date_naissance" type="dateTime"/>
        <xs:element name="adresse">
          <xs:sequence>
            <xs:complexType>
              <xs:element name="rue" type="xs:string"/>
              <xs:element name="ville" type="xs:string"/>
            </xs:complexType>
          </xs:sequence>
        </xs:element>
    </xs:sequence>
  </xs:element>
</xs:schema>
```

L'ensemble des schémas gérés étant décrit, il s'agit ensuite d'énumérer l'ensemble de chemins (*pathset*) associés à chaque collection, et de nommer chacune des collections.

On définit ainsi la collection `LIVRES`.

```
  </schemas>

  <collections>
    <collection name="LIVRES" ns="www.bibliographie.org">
    <step name="livre">
      <step name="titre"/>
      <step name="numero"/>
      <step name="date"/>
      <step name="auteur">
        <step name="nom"/>
        <step name="prenom"/>
      </step>
    </step>
    </collection>
```

Et la collection PERSONNES.

```
    <collection name="PERSONNES" ns="http://www.qui-es-tu.org">
    <step name="personne">
      <step name="nom"/>
      <step name="prenom"/>
      <step name="date_naissance"/>
      <step name="adresse">
        <step name="rue"/>
        <step name="ville"/>
      </step>
    </step>
  </collection>
</collections>
</xm:metadata>
```

Nous avons ainsi entièrement donné la description de métadonnées dans un fichier XML où la première partie de description de métadonnées comporte l'ensemble des XML-Schema gérés par cette source et la seconde partie de description de métadonnées détermine sous quels noms de collections sont gérées les structures des données. La figure 3.4 (b) représente la description de métadonnées que nous venons de décrire.

Le médiateur ayant interrogé tous les adaptateurs -auxquels il a accès- sur leurs schémas respectifs, il fusionne ensuite les descriptions de métadonnées sous forme d'une description de métadonnées générale comportant les schémas et leurs caractéristiques (adaptateur associé, nom de la collection).

Soit le médiateur M2 comportant les adaptateurs $W2$, $W3$ et $M3$. La source gérée par W2 (figure 3.4 (a)) contient les collections MAGASIN et ARTICLES toutes deux issues du schéma http://magasin.general.com/commandes. La source gérée par W3 (figure 3.4 (b)) contient les collections PERSONNES et LIVRES issues respectivement des schémas http://www.qui-es-tu.org/identite et http://www.bibliographie.org/bibliotheque. Enfin la source gérée par M3 (figure 3.4 (c)) contient les collections RESTAURANTS et BARS issues du schéma http://www.sortir-a-paris.fr/, HOTELS issue du schéma http://bon-voyage.com/reservation, et les collections LIVRES et MAGASINS issus des schémas déjà connus http://www.bibliographie.org/bibliotheque et http://magasin.general.com/commandes.

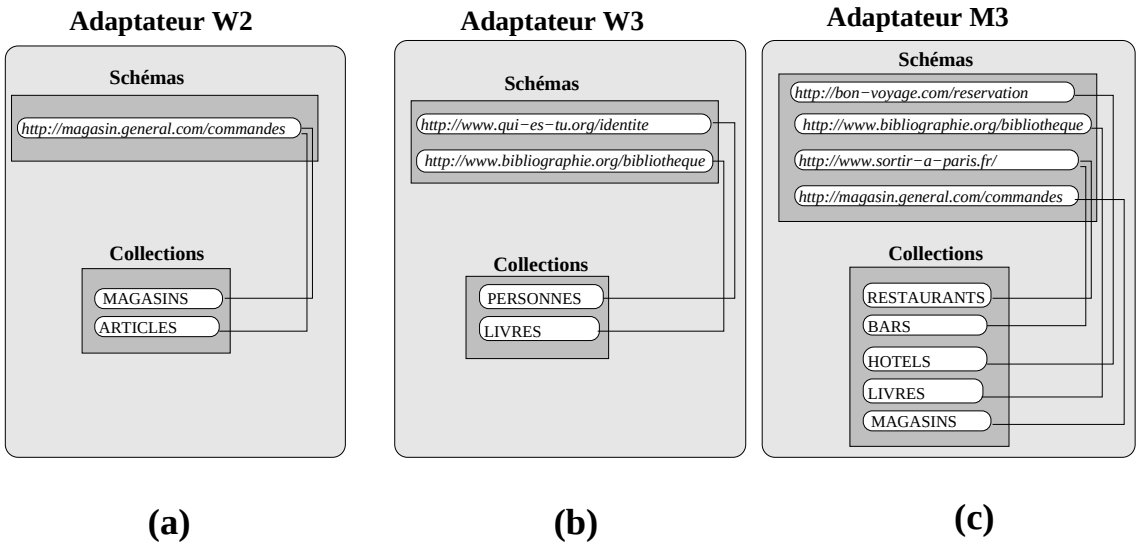


FIG. 3.4 – description de métadonnées des sources gérées par W2, W3 et M3

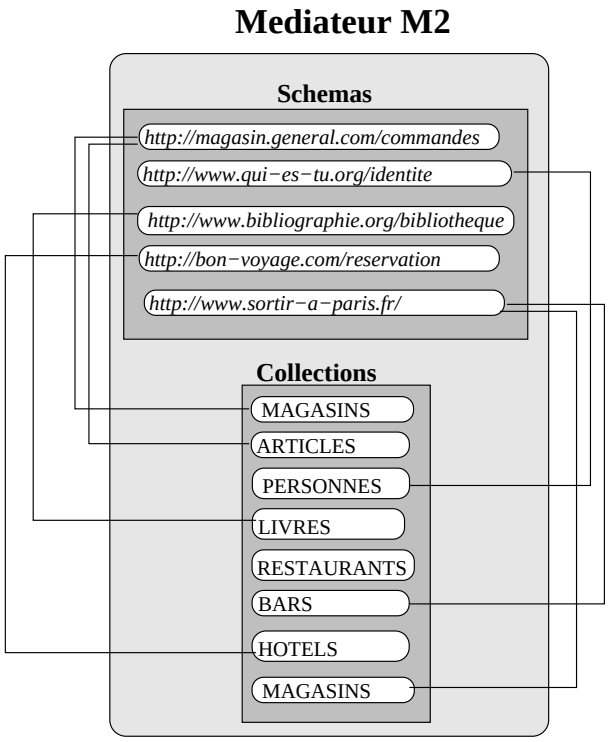


FIG. 3.5 – Description de métadonnées du médiateur M2

Le contenu du cache de la description de métadonnées du médiateur M2 est représenté par la figure 3.5. L'ensemble des schémas de cette description de métadonnées correspond à l'union des schémas de W2, W3 et M3, et l'ensemble des collections à l'union des col-

lections de W2, W3 et M3.

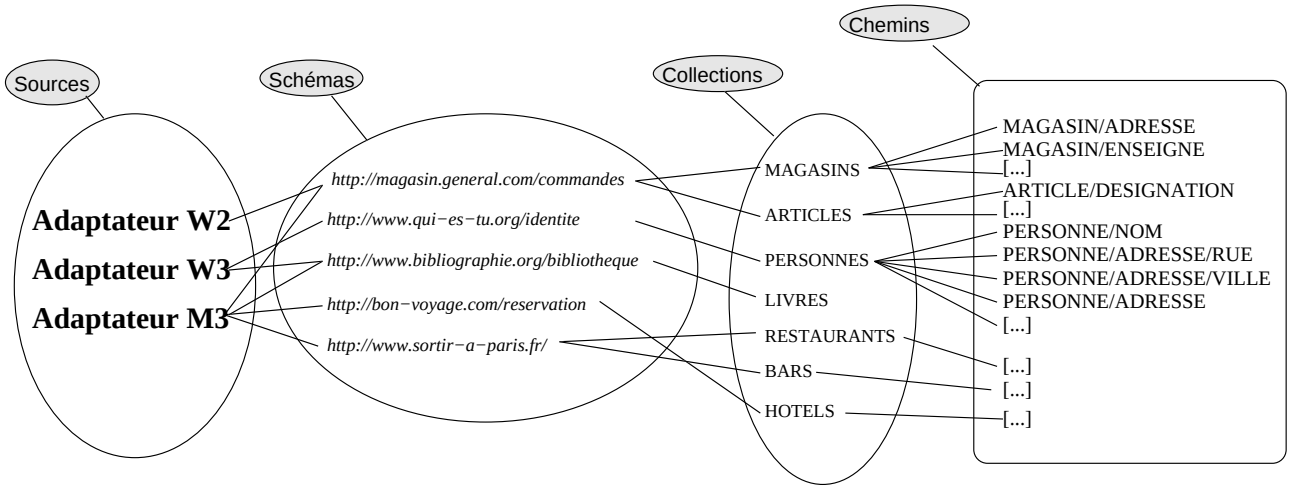


FIG. 3.6 – Description de métadonnées du médiateur M2

Pour pouvoir répondre rapidement aux demandes de typage de données, il y a plusieurs index (cf. figure 3.6) associés à l'ensemble des schémas :

- *index de source* : à partir d'un nom d'adaptateur, il est possible de retrouver tous les schémas gérés par cet adaptateur ;
- *index de schéma* : à partir d'un espace de noms (*namespace*) associé à un schéma, il est possible de retrouver les adaptateurs qui le gère. À partir d'un espace de noms, il est également possible d'obtenir toutes les collections l'utilisant ;
- *index de collection* : à partir d'un nom de collection, il est possible de retrouver tous les schémas associés à ce nom de collection. On peut également générer tous les chemins associés à cette collection ;
- *index de chemins* : à partir d'un chemin donné, il est possible de retrouver les collections les gérant.

Les index peuvent être utilisés conjointement, il est ainsi possible d'utiliser les critères de plusieurs de ces index afin de retrouver un sous-schéma particulier. Par exemple, on peut demander précisément le sous-schéma de tous les nœuds de la source gérée par W1 dont la collection s'appelle PERSONNES et dont le chemin est PERSONNE/ADRESSE, ou alors, plus vaguement, demander tous les schémas de la source gérée par W1, ou encore, tous les sous-schémas dont le chemin est PERSONNE/ADRESSE, sans préciser ni de noms de collection, ni de noms d'adaptateur.

3.5.2.2 Exportation de capacité des sources

Les adaptateurs reliés aux médiateurs sont spécifiques à la source qu'ils gèrent. Les sources sont hétérogènes et peuvent être interrogées de façon plus ou moins complexe. Ainsi, un SGBD relationnel fournit une interface avancée, permettant des requêtes complexes d'interrogation, de création et de mises à jour. Une page web au contraire, est une source avec une interface très pauvre ne permettant que des requêtes très limitées.

Des architectures de médiations comme TSIMMIS [Li *et al.* 1998] et DISCO [Tomasic *et al.* 1996] ont mis au point un langage très précis d'exportation des capacités de la source. Il est ainsi possible de spécifier des capacités très précises. Par exemple, il est possible de préciser que l'on ne peut sélectionner que les tuples d'une relation `Pub1` lorsque l'attribut `author` est connu.

Lorsqu'un nouvel adaptateur est enregistré auprès du médiateur, ce dernier lui demande les informations de capacité de la source *via* une méthode de l'API XML/DBC `getCapacity ()`.

Liste de règles de capacités Une *liste de règles de capacités* d'un adaptateur est composée d'une liste de *règles* numérotées de 1 à 65535. Celles-ci servent à la vérification de la capacité d'un adaptateur à répondre à une requête. L'expression est alors comparée *successivement* à chaque règle de la *liste de règles de capacités*. Lorsque une correspondance est trouvée, la réponse correspondant à la règle trouvée est alors renvoyée. Cette réponse peut être soit *allow*, ce qui veut dire que la requête peut être pris en charge par l'adaptateur, soit *deny* dans le cas contraire.

Une *règle de capacité* se définit par au plus 8 champs appelés *propriétés*.

1. *Le numéro de règle (champ p_1)* : les règles sont *ordonnées* suivant leur numéro de règles, et lues dans cet ordre lors de la vérification des règles.
2. *La permission (champ p_2)* : positionnée soit à *allow* soit à *deny* indique si la requête donnée peut être pris en charge par l'adaptateur ou non.
3. *L'opérateur algébrique (champ p_3)* : l'opérateur algébrique sur laquelle s'applique la règle (*scan, project, join, select*).
4. *Le nom de collection (champ p_4)* : le nom de la collection sur laquelle s'applique l'opérateur algébrique (valable pour *scan, project, select, join*).
5. *Le chemin (champ p_5)* : le chemin (ou l'attribut dans le cas du relationnel) sur lequel s'applique l'opérateur.
6. *L'opérateur de comparaison (champ p_6)* : $<, >, \leq, \geq, =, ==$ utilisé par l'opérateur (valable pour *join* et *select*).
7. *Le nom de collection 2 (champ p_7)* : la collection avec laquelle la jointure est possible (valable pour *join*).

8. *Le chemin (champ p_8)* : le chemin (ou l'attribut) de collection avec lequel la jointure est possible (valable pour *join*).

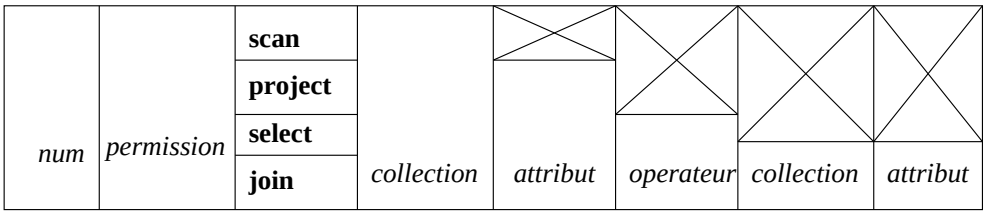


FIG. 3.7 – Règle d'exportation des capacités de la source

Dans une règle, seuls les mots-clefs *allow* et *deny* sont obligatoires, les autres arguments sont facultatifs (mais l'ordre et le type des arguments si ceux-ci sont existants, doivent respecter la syntaxe décrite dans le tableau de la figure 3.7. Lorsqu'un argument n'est pas précisé, on assume que tous les éléments correspondent.

Une *liste de règles de capacités* inclus toujours une règle numérotée 65535. Cette règle numérotée 65535 due à la notion de d'ordonnancement des règles est la dernière règle de la liste. Cette règle doit impérativement être soit *allow* soit *deny* et ne pas comporter d'autres champs. De cette façon, la dernière règle correspondra à toutes les requêtes qui n'auraient pas été traitées par les règles précédentes. On l'appelle *règle par défaut.*

On notera que si la règle par défaut est *allow*, cela voudra dire que toutes les opérations non-explicitement définie dans les règles précédentes comme interdite seront autorisée. Cette politique *permissive* correspond à un SGBD relationnel ou tout système de gestion de bases de données suffisamment complexe pour implémenter la plupart des opérateurs.

À l'inverse, si la règle par défaut est *deny*, toute requête non explicitement autorisée est interdite. Cette politique *restrictive* correspond à un moteur de recherche ou un système de données très simple n'ayant qu'un système de requête limité.

Ci-dessous, un exemple, une *liste de règles de capacités* :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
10	allow	scan					
20	allow	select	personne				
30	allow	select	voiture		=		
100	allow	select	voiture	age	<		
200	allow	project	personne	nom			
260	allow	project	voiture				
261	allow	join	personne	id	=	voiture	id_conducteur
65535	deny						

Cette *liste de règles de capacités* se décrit de la façon suivante :

- *règle 10* : autoriser la lecture séquentielle pour toutes les collections ;
- *règle 20* : autoriser toute sélection sur la collection *personne* ;
- *règle 30* : autoriser la restriction sur la collection *voiture* si elle fait intervenir l'opération d'égalité ;
- *règle 100* : autoriser la restriction sur l'attribut de la collection *voiture* si elle fait intervenir l'opération d'infériorité ;
- *règle 200* : autoriser la projection sur l'attribut *nom* de *personne* ;
- *règle 260* : autoriser la projection sur n'importe quel attribut de voiture ;
- *règle 261* : autoriser l'équi-jointure entre l'attribut *id* de *personne* et l'attribut *id_conducteur* de *voiture* ;
- *règle 65535* : interdire toutes les autres opérations. Comme elle englobe tous les opérateurs, c'est la règle par défaut (politique restrictive).

Un exemple d'une *liste de règles de capacités* de type permissif :

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
1	allow	select	voiture		$<$		
2	deny	select					
65535	allow						

On autorise toutes les opérations sauf les sélections, mais les sélections portant sur la collection *voiture* si elles font intervenir l'opérateur $<$ sont autorisées.

Simplification des règles de capacités Dûe à l'ordonnancement des règles et au caractère générique des propriétés non renseignées il est possible de simplifier une *liste de règles de capacités*.

Une règle plus générique qu'une autre sur les mêmes propriétés est superflue si elle est placé avant. Ainsi par exemple :

```
1   deny  select
2   allow select voiture
```

Dans cette liste de règles de capacités, toutes les sélections vérifient la règle 1 et ne passeront jamais par la règle 2. La règle 2 est donc inutile et peut être supprimée.

On peut formaliser cela de la façon suivante :

Soit une règle de capacité R_1 dont les colonnes sont $p_1, p_2, ...p_n$. Soit une règle de capacité R_2 dont les colonnes sont $p'_1, p'_2, ...p'_n$. Si R_1 précède R_2, *ie.* si $p_1 < p'_1$, et que $\forall i \in [3-n]$, si $p_i = \emptyset$ *ou que* $p_i = p'_i$, alors la règle R_2 est inutile quelque soit sa permission p'_2.

L'inverse n'est pas vrai : si une règle R_1 est plus générique qu'une autre R_2, et qu'elle est placée après ET si les actions (allow/deny) sont contraires, les règles sont toutes les deux utiles. Dans ce cas, la règle R_2 est une *exception* de la règle R_1.

Par exemple :

```
1   allow select voiture
2   deny  select
```

Dans cette liste de règles de capacités, tous les *select* sont interdits sauf ceux sur *voiture*.

Si une règle R_1 est plus générique qu'une autre R_2, et qu'elle est placée après ET si les actions sont les mêmes, la règle R_2 est redondante et inutile.

Exportation des règles de capacité Les règles de capacité que nous avons décrites sont exportées par les adaptateurs sous forme de document XML. Ce document XML doit être conforme au schéma décrit dans l'annexe B.

Ainsi, l'exemple de liste de règles de capacités décrite au début de la section se traduit par le document suivant :

```
<ruleset>
    <rule num="10">
        <permission> allow </permission>
        <relationalop> scan </relationalop>
    </rule>
    <rule num="20">
        <permission> allow </permission>
        <relationalop> select </relationalop>
        <collection1> personne </collection1>
    </rule>
    <rule num="30">
        <permission> allow </permission>
        <relationalop> select </relationalop>
        <collection1> voiture </collection1>
        <operator> equal </operator>
    </rule>
    <rule num="100">
        <permission> allow </permission>
        <relationalop> select </relationalop>
        <collection1> voiture </collection1>
        <attribute1> age </attribute1>
        <operator> less </operator>
    </rule>
    <rule num="200">
        <permission> allow </permission>
        <relationalop> project </relationalop>
        <collection1> personne </collection1>
        <attribute1> nom </attribute1>
    </rule>
    <rule num="260">
        <permission> allow </permission>
        <relationalop> project </relationalop>
        <collection1> voiture </collection1>
    </rule>
    <rule num="261">
        <permission> allow </permission>
        <relationalop> join </relationalop>
        <collection1> personne </collection1>
        <attribute1> id </attribute1>
        <operator> equal </operator>
        <collection2> voiture </collection2>
        <attribute2> id_conducteur </attribute2>
    </rule>
    <rule num="65535">
        <permission> deny </permission>
    </rule>
</ruleset>
```

Prise en compte des capacités des adaptateurs par le médiateur Avant de passer une sous-requête atomique (voir décomposition des requêtes en requête atomique chapitre 3) à un adaptateur, on vérifie si celle-ci peut être prise en charge par l'adaptateur. Pour cela on applique l'algorithme suivant :

```
pour chaque requête simple R à appliquer sur un adaptateur A
       appliquer la liste des règles de capacité de l'adaptateur A.
        si l'opérateur est accepté
                alors la sous-requête peut être envoyée à l'adaptateur.
        sinon
               il devra être pris en charge par le médiateur.
        fin si
fin pour
```

3.5.2.3 Exportation de statistiques et formules de coût des sources

Chaque adaptateur peut avoir des formules de coût et des statistiques qui lui sont spécifiques. Dans ce cas, il faut pouvoir intégrer ces formules de coût dans le coût global du médiateur.

Lorsqu'un nouvel adaptateur est enregistré auprès du médiateur, ce dernier lui demande les informations de coût de la source *via* une méthode de l'API XML/DBC `getCost ()`.

Afin que les adaptateurs puissent communiquer leurs informations au médiateur, nous définissons à l'instar de DISCO [Tomasic *et al.* 1997] un langage permettant aux adaptateurs d'exporter leur modèle de coût. Le langage s'inspire de celui défini par DISCO, quelques modifications ayant été apportées afin d'intégrer les aspects de parcours de chemin propres aux données semi-structurées. L'originalité de notre langage est qu'il est défini en XML.

Un langage d'expressions mathématiques basé sur XML : MathML Nous nous intéressons à la manière de définir des variables et différentes formules de coûts par un langage basé sur XML. Ce langage doit être suffisamment général pour décrire n'importe quelle formule mathématique.

MathML [Ausbrooks *et al.* 2002] est une spécification du W3C qui permet de coder en XML à la fois la *représentation* d'un objet mathématique (notation) mais aussi la *structure mathématique* de l'objet même (le contenu, l'idée).

Par exemple, la *représentation* de la *structure mathématique* qu'est la division de x par y peut être soit $x \div y$ soit $\frac{x}{y}$.

Nous nous intéresserons qu'à la structure mathématique (*MathML Content*).

Une expression mathématique est composée d'opérateurs, d'identifiants et de valeurs. L'élément *cn* permet de représenter des nombres entiers, rationnels, réels ou complexes. L'élément *ci* est utilisé pour représenter tous les identifiants. Par exemple, les variables d'une fonction. L'élément *apply* permet de grouper les opérateurs avec argu-

ments. L'élément *declare* permet de définir une fonction ou une variable. Pour cela, on introduit l'identifiant de la nouvelles fonction ou variable par l'élément *ci*, et si c'est une fonction paramétrée les variables avec les éléments *lambda* et *bvar*. Enfin il existe pour la plupart des opérateurs ou fonctions, un élément le désignant. Le tableau suivant donne un aperçu des éléments définis en MathML 2.0.

algèbre logique	
quotient	modulo
factorial	factoriel
divide	division
plus	addition
minus	soustraction
times	multiplication
min	minimum
max	maximum
and	« et » logique
...	...
calculs	
int	intégrale
diff	dérivation
partialdiff	dérivation partielle
...	...
fonctions classiques	
ln	logarithme néperien
exp	exponentielle
cos	cosinus
...	...

statistiques	
mean	moyenne
sdev	déviation standard
variance	variance
...	...
constantes et symboles	
integers	l'ensemble des entiers
complexes	l'ensemble des nombres complexes
pi	$\pi = 3,1415...$
eulergamma	constante d'Euler
...	...
algèbre linéaire	
vector	vecteur
matrix	matrice
determinant	déterminant d'une matrice
...	...
séquences et séries	
sum	somme des termes d'une séquence
product	produit des termes d'une séquence
limit	limite vers laquelle tend une séquence
...	...
...	

La figure 3.8 montre comment exprimer l'expression (a) sous forme MathML (c). (b) montre la décomposition intermédiaire de l'expression (a) sous forme d'arbre algébrique.

En utilisant MathML, on donne la possibilité de définir une fonction, d'affecter des variables et de réaliser des appels de fonctions et de variables.

Par exemple :

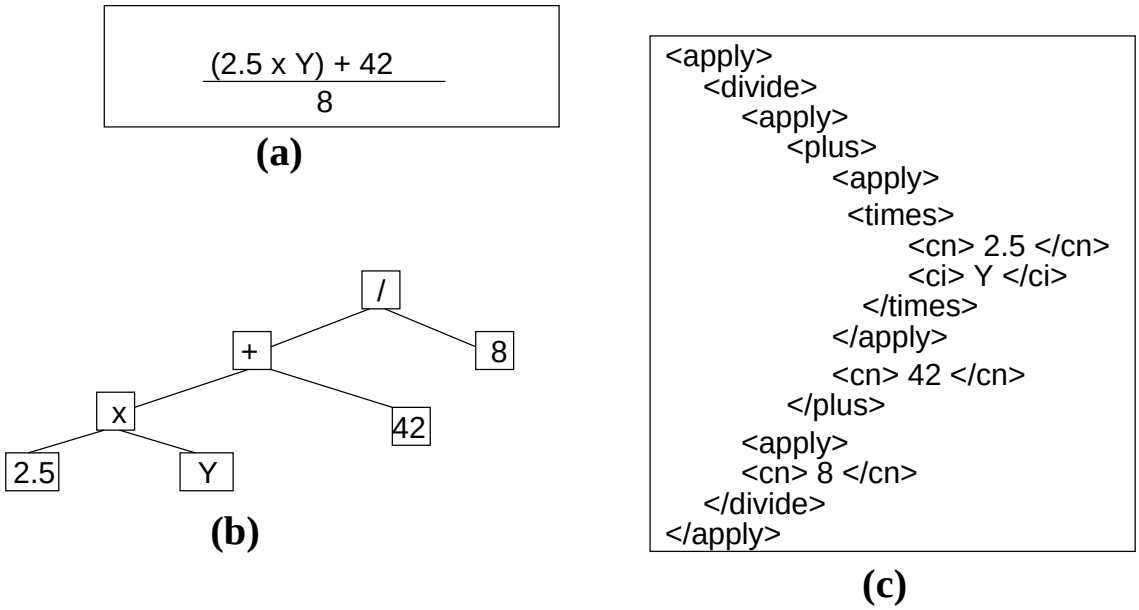


FIG. 3.8 – Expression mathématique en MathML

```
(0)    <declare type="real">
           <ci> CPU </ci>
           <cn> 0.1 </cn>
       </declare>

(1)    <declare type="real">
           <ci> ES </ci>
           <cn> 0.4 </cn>
       </declare>

(2)    <declare type="real">
           <ci> MA_VAR1 </ci>
           <cn> 42 </cn>
       </declare>
```

Dans (0)(1)(2) on affecte (`+<declare/>+`)dans des variables que l'on nomme `CPU`, `ES` et `MA_VAR1` respectivement les valeurs 0.1, 0.4 et 42. Soit en pseudo-code :

```
CPU := 0.1
ES := 0.4
MA_VAR1 := 42
```

```
(3)    <declare type="fn">
           <ci> ma_fonction </ci>
           <lambda>
               <bvar>
                   <ci> x </ci>
                   <ci> y </ci>
               </bvar>
               <apply>
                   <minus>
                       <apply>
                           <plus>
                               <ci> x </ci>
                               <ci> CPU </ci>
                               <ci> y </ci>
                           </plus>
                       </apply>
                       <ci> 8 </ci>
                   </minus>
               </apply>
           </lambda>
       </declare>
```

Dans (3) on définit ensuite la fonction `ma_fonction` qui additionne son premier argument avec la valeur de la variable `CPU` avec son second argument et qui retranche 8. Soit en pseudo-code :

```
ma_fonction (x, y) = (x + CPU + y ) - 8
```

```
(4)     <declare type="real">
            <ci> ma_var2 </ci>
            <apply>
                <plus>
                    <ci> MA_VAR1 </ci>
                    <apply>
                        <ci> ma_fonction </ci>
                        <ci> ES </ci>
                        <cn> 36 </cn>
                    </apply>
                </plus>
            </apply>
        </declare>
```

Dans (4), on affecte à la variable `ma_var2` la valeur de la variable `MA_VAR1` additionnée au retour de la fonction `ma_fonction` appelée avec comme arguments la valeur de la variable `ES` et la valeur 36. Soit en pseudo-code :

```
ma_var2 := MA_VAR1 + ma_fonction (ES, 36)
```

```
(5) <reln>
        <eq>
            <apply>
               <ci> ma_fonction_interne </ci>
               <cn> LIVRE </ci>
               <cn> 26 </cn>
             </apply>
             <cn> 327 </cn>
        </eq>
    </reln>
```

Et enfin dans (5) il s'agit d'une fonction interne qui **si** elle est appelée avec **exactement** les paramètres précisés (ici "LIVRE"et 26), vaut 327. Soit en pseudo-code :

```
ma_fonction_interne ("LIVRE", 26) = 327
```

Les noms des variables et des fonctions sont arbitraires et sont définis au fur et à mesure de leurs déclarations.

Certains noms de variables et de fonctions ont néanmoins une signification particulière pour l'évaluateur de coût, elles peuvent représenter des statistiques systèmes, des statistiques de données ou des formules de coût d'opérateurs de notre algèbre. Ces variables et fonctions prédéfinies seront détaillées dans le chapitre 5. Il faut bien garder à l'esprit que ces variables ne sont utilisées que dans des fonctions et primitives définies plus loin dans le code d'exportation, soit pour un nombre très limité. Les fonctions d'opérateurs ne sont pas définies dans le code d'exportation, comme valeur par défaut des fonctions génériques pour ces opérateurs.

On distingue les statistiques systèmes, les statistiques de collections et les formules de coût. L'exportation sous la forme d'un document XML respectant le schéma donné dans l'annexe A et se présentant sous la forme suivante : suivant :

```
1       <costmodel>
2           <statistics>
3               <system>
4                   expression(s) mathématique(s)
5               </system>
6
7               <collection>
8                   expression(s) mathématique(s)
9               </collection>
10
11              <user-defined>
12                  expression(s) mathématique(s)
13              </user-defined>
14          </statistics>
15
16          <formulas>
17              <user-defined>
18                  expression(s) mathématique(s)
19              </user-defined>
20
21              <generic>
22                  expression(s) mathématique(s)
23              </generic>
24
25              <operators>
26                  expression(s) mathématique(s)
27              </operators>
28          </formulas>
29      </costmodel>
```

Les expressions mathématiques utilisent MathML et sont définies entre les balises : `<math xmlns="http://www.w3.org/1998/Math/MathML">` et `</math>`.

On définit en ligne (4) les statistiques du système, (8) les statistiques de collection, (12) des variables intermédiaires définies par l'utilisateur, (18) les fonctions intermédiaires définies par l'utilisateur, (22) les formules de coût génériques pour les temps à chaud, temps à froid et enfin en (26) les formules de coût associées aux opérateurs algébriques (scan, project, etc.)

3.5.3 Exécution de requêtes

L'exécution de requêtes sur les sources se fait par l'intermédiaire de la méthode de l'interface XML/DBC `executeQuery (requete)`. Les requêtes sont exprimées dans le langage XQuery. Nous donnons ci-dessous quelques exemples de requêtes sur la base.

Exemple : *Renvoyer le nom de l'auteur et le titre des livres dont la date est postérieure à 1985 et dont le titre contient le mot « UNIX »*

```
for $l in Collection("*:*")/livre
where
       $l/date < 1985
    and
       contains($l/titre, "UNIX")
return
    <livre>
        <auteur>$l/auteur/nom</auteur>
        <titre>$l/titre</titre>
    <livre>
```

La réponse à la requête se présente sous forme d'un ensemble de documents XML (un par résultat trouvé), accessibles par un curseur se déplaçant dans l'ensemble des résultats par les méthodes `next ()` et générant un document XML (`getAsDocument ()`) ou un flux SAX (`getAsSAX ()`) à analyser.

Le résultat de la requête ci-dessus sera :

```
<results>
    <livre>
        <auteur>Bach</auteur>
        <titre>The design of the UNIX Operating System</titre>
    <livre>
    <livre>
        <auteur>Stevens</auteur>
        <titre>UNIX Network Programming</titre>
    <livre>
    <livre>
        <auteur>Moreno</auteur>
        <titre>UNIX administration</titre>
    <livre>
    <livre>
        <auteur>Rifflet</auteur>
        <titre>La programmation sous UNIX</titre>
    <livre>
</results>
```

3.6 Plan d'exécution

Il peut y avoir plusieurs, voire une infinité de manières de traiter une requête ; chacune de ces façons étant appelée un *plan d'exécution*. L'ensemble des *plans d'exécution équivalents* (c'est-à-dire donnant le même résultat pour une requête donnée) forme *l'espace des possibilités*. La cardinalité de cet ensemble pouvant être très grande, il est impossible d'explorer *toutes* les possibilités afin de déterminer le *plan optimal*. Il s'agit donc de proposer des heuristiques afin de restreindre cet ensemble en un ensemble fini beaucoup plus petit appelé *espace de recherche*.

Le rôle de l'*optimiseur* est de déterminer l'espace de recherche, puis d'examiner chacune des possibilités d'exécution afin de choisir le plan optimal dans cet espace.

Le *coût* d'un modèle d'exécution peut s'exprimer en terme de *temps d'exécution* (temps observé entre le lancement de la requête jusqu'à l'obtention des résultats), de *travail* (consommation de ressources : communications, place mémoire), ou encore d'*unités monétaires* (prix des requêtes et des communications).

La manipulation de sources hétérogènes implique des traitements différents selon les sources. De ce fait, les algorithmes classiques d'optimisation utilisés dans les bases de données ne peuvent pas tous s'appliquer dans le cas d'une optimisation sur des sources de données hétérogènes, ceci à cause des mauvaises voire l'absence de connaissances des propriétés des données manipulées (index, distribution, schémas, cardinalités).

D'autres choix deviennent essentiels : l'ordonnancement, le regroupement des sous-requêtes et la parallélisation des requêtes vers les sources suivant leurs capacités.

La construction simple d'un plan d'exécution se fait suivant les étapes suivantes :

1. *Normalisation et canonisation de la requête* : il s'agit de transformer la requête en suivant certaines règles d'équivalence afin d'obtenir une forme générique plus apte à être traitée dans la suite de nos opérations.
2. *Atomisation des requêtes* : il s'agit de décomposer la requête en identifiant les différents ensembles de sources.
3. *Identification des dépendances et création de l'arbre de dépendance* : il s'agit de déterminer les dépendances entre les différents atomes de la requête.
4. *Identification des sources* : à l'aide de cache de descriptions de métadonnées on peut associer les adaptateurs correspondants au type de source demandé.
5. *Création du plan d'exécution* : L'arbre de dépendance est ensuite utilisé afin de générer un *plan d'exécution.*
6. *Optimisation du plan d'exécution* : il s'agit de générer des plans d'exécution équivalents suivant des stratégies d'ordonnancement des opérateurs, et de choisir celui de moindre coût.

Dans la section suivante, nous allons détailler le principe et les algorithmes utilisés pour chaque étape citée ci-dessus. Afin de faciliter la compréhension de chacune des étapes, nous allons dérouler au fur et à mesure le traitement appliquée à l'évaluation de la requête exemple donnée ci-dessous.

Requête exemple XQuery : *(Q-1.) Afficher pour chaque hôtel de catégorie 3 étoiles, les clients y ayant séjourné et qui ont écrit un livre dont le titre comporte le mot « UNIX ».*

```
for $h  in Collection ("*")/hotel
where
   $h/categorie = "3"
return
   <hotel>
       <nom>$h/nom</nom>
       <auteurs>
            for $p  in Collection ("*")/personne
            for $l  in Collection ("*")/livre
            where
               $p/nom = $l/auteur/nom
                and contains ($l/titre, "UNIX")
                and $p/nom = $h//client
            return
               <auteur>
                   <prenom>$p/prenom</prenom>
                   <nom>$p/nom</nom>
                   <livre>$l/titre</livre>
               <auteur>
       </auteurs>
   </hotel>
```

La requête est appliquée au médiateur M2 de l'architecture de la figure figure 3.2. Ses descriptions de métadonnées ont été décrites à la figure 3.5.

Pour pouvoir suivre l'exemple, nous donnons en plus le schéma de `http ://www.bon-voyage.com/reservation` :

```
   <xs:schema xmlns="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://www.bon-voyage.com/reservation"
       xmlns="http://www.bon-voyage.com/reservation">
     <xs:element name="hotel">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="nom" type="xs:string"/>
           <xs:element name="adresse" type="xs:string"/>
           <xs:element name="clientele">
             <xs:sequence>
               <xs:complexType>
                 <xs:element name="client" type="xs:string"/>
                 <xs:element name="dateDArrivee" type="xs:string"/>
                 <xs:element name="dateDeDepart" type="xs:string"/>
               </xs:complexType>
             </xs:sequence>
           </xs:element>
       </xs:sequence>
     </xs:element>
   </xs:schema>
```

et la définition de la collection "HOTELS" :

```
<collection name="HOTELS" ns="http://www.bon-voyage.com/reservation">
<step name="hotel">
  <step name="nom"/>
  <step name="adresse"/>
  <step name="clientele">
    <step name="client"/>
    <step name="dateDArrivee"/>
    <step name="dateDeDepart"/>
  </step>
</step>
</collection>
```

3.7 Traitement d'une requête XQuery

Nous nous sommes appuyés sur des règles de normalisation du langage XQuery. En effet, une requête XQuery, peut comporter des requêtes imbriquées (*nested queries*) difficiles à évaluer, pour cela, il est utile de « désimbriquer » ces requêtes afin d'avoir une requête plus simple à traiter. Ces transformations sont faites en suivant des règles d'équivalence très semblables à celles détaillées dans l'article de [Manolescu *et al.* 2001]. Nous adopterons les notations suivantes, les lettres minuscules x, y, z désigneront des variables XQuery, et les lettres capitales E, C, R désigneront des requêtes XQuery. Pour raisons de simplicité, on abrégera des requêtes de la forme
« **for** x_1 **in** E_1, x_2 **in** E_2, **...,** x_n **in** E_n»
en
« **for** $\overrightarrow{x}$ **in** E ».
Dans ce cas, E est une expression d'arité n et $\overrightarrow{x}$ est relié consécutivement à chaque tuple de valeur d'évaluation de E.

Les clauses `let` peuvent être considérées comme des variables temporaires de définitions. Lors de la normalisation elles sont éliminées par la règle suivante :

for $\overrightarrow{x}$ **in** E_1 **let** $\overrightarrow{y} = E_2(\overrightarrow{x})$ **for** $\overrightarrow{z}$ **in** $E_3(\overrightarrow{x}, \overrightarrow{y})$ **where** $C(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{z})$ **return** $R(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{z})$	$\Leftrightarrow$	**for** $\overrightarrow{x}$ **in** E_1, $\overrightarrow{z}$ **in** $E3(\overrightarrow{x}, E_2(\overrightarrow{x}))$ **where** $C(\overrightarrow{x}, E_2(\overrightarrow{x}), \overrightarrow{z})$ **return** $R(\overrightarrow{x}, E_2(\overrightarrow{x}), \overrightarrow{z})$

3.7.1 Canonisation des requêtes

À cette règle de normalisation simples, on y ajoute une procédure de *canonisation* consistant à séparer les expressions de la requête de la reconstruction. Pour cela, on sépare les blocs de requêtes XQuery en plusieurs sous-requêtes *sans tenir compte de la*

reconstruction que l'on nommera QDB et une requête de reconstruction appelée QMem. Cette décomposition est effectuée suivant les règles ci-dessous :

for $\overrightarrow{x}$ **in** E_1 **where** $C_1(\overrightarrow{x})$ **return** $R(\overrightarrow{x})$	$\Rightarrow$	**let** $t_1 :=$ **for** $\overrightarrow{v_1}$ **in** E_1 **where** $C_1(\overrightarrow{v_1})$ **return** $(\overrightarrow{v_1})$	QDB_1
		$R(\overrightarrow{v_1})$	$QMem$

où $\overrightarrow{v_1} = \overrightarrow{x}$

et

for $\overrightarrow{x}$ **in** E_1 **where** $C_1(\overrightarrow{x})$ **return** $R($ **for** $\overrightarrow{y}$ **in** E_2 **where** $C_2(\overrightarrow{x}, \overrightarrow{y})$ **return** $R(\overrightarrow{x}, \overrightarrow{y}))$	$\Rightarrow$	**let** $t_1 :=$ **for** $\overrightarrow{v_1}$ **in** E_1 **where** $C_1(\overrightarrow{v_1})$ **return** $\overrightarrow{v_1}$	QDB_2
		let $t_2 :=$ **for** $\overrightarrow{v_2}$ **in** t_1 **for** $\overrightarrow{v_3}$ **in** E_2 **where** $C_2(\overrightarrow{v_2}, \overrightarrow{v_3})$ **return** $(\overrightarrow{v_2}, \overrightarrow{v_3})$	QDB_2
		$R(\overrightarrow{v_1}, \overrightarrow{v_2}, \overrightarrow{v_3})$	$QMem$

où

$$\begin{cases} \overrightarrow{v_1} = \overrightarrow{x} \\ \overrightarrow{v_2} = \overrightarrow{x} \cap \overrightarrow{y} \\ \overrightarrow{v_3} = \overrightarrow{y} \setminus \overrightarrow{x} \end{cases}$$

L'ensemble des règles de normalisation et de canonisation sont internes aux travaux effectués à la société e-XMLMedia et ne peuvent être entièrement dévoilées dans cette thèse. Elles ne font de toute façon pas l'objet de cette thèse.

Définition 3.1 : normalisation d'une requête

La *normalisation d'une requête* consiste à supprimer autant que possible les imbrications.

Définition 3.2 : canonisation d'une requête

La *canonisation d'une requête* consiste à regrouper les différents blocs d'évaluation de la requête en sous-requêtes nommées QDB ne comportant pas de reconstruction, et en générant une requête de reconstruction appelée QMem.

Dans l'exemple de la requête Q-1, on distingue deux blocs de construction. Le premier concerne l'ensemble des hôtels, le second concerne pour chacun des hotels la liste des `livres/personnes` associés. On génère donc les deux QDB et la QMem suivants :

let $t_1 :=$ **for** $\$h$ **in** $Collection(" * ")/hotel$ **where** $\$h/categorie = "3"$ **return** $(\$h/nom, \$h//client)$	QDB_1
let $t_2 :=$ **for** $\$t$ **in** $\$t_1$ **for** $\$p$ **in** $Collection(" * ")/personne$ **for** $\$l$ **in** $Collection(" * ")/livre$ **where** $\$p/nom = \$l/auteur/nom$ **and** $contains(\$l/titre, "UNIX")$ **and** $\$p/nom = \$t//client$ **return** $(\$p/prenom, \$p/nom, \$l/titre)$	QDB_2
$< hotel >$ $< nom > \$t_1/nom < /nom >$ $< auteurs >$ $< auteur >$ $< prenom > \$p/prenom < /prenom >$ $< nom > \$p/nom < /nom >$ $< livre > \$l/titre < /livre >$ $< auteur >$ $< /auteurs >$ $< /hotel >$	$QMem$

3.7.2 Atomisation des requêtes

Une fois la requête canonisée, il s'agit ensuite de déterminer dans chacune des sous-requêtes QDB, les différents ensembles de collections de documents. Ces différents ensembles identifiés sont appelés *requête atomique* et sont notés QA_i. La décomposition en requêtes atomiques se fait suivant la règle suivante :

for $\overrightarrow{x}$ **in** E_1 **where** $C(\overrightarrow{x})$ **return** $(\overrightarrow{x})$	$\Rightarrow$	**for** x_1 **in** $E_1(x_1)$ **where** $C_{x_1}(x_1)$ **return** $(\overrightarrow{x_1})$	QA_1
		for x_2 **in** $E_2(x_2)$ **where** $C_{x_2}(x_2)$ **return** $(\overrightarrow{x_2})$	QA_2
		...	
		for x_n **in** $E_n(x_n)$ **where** $C_{x_n}(x_n)$ **return** $(\overrightarrow{x_n})$	QA_n
		where $C(\overrightarrow{x_1}, \overrightarrow{x_2}, ..., \overrightarrow{x_n})$	$QDEP$

Définition 3.3 : atomisation d'une requête

L'*atomisation d'une requête* consiste à séparer la requête en sous-requêtes portant chacune sur une seule collection de documents et une *requête de liaison*. Les sous-requêtes sont appelées *requêtes atomiques*.

Dans l'exemple de la requête Q-1 que nous avions canonisé, QDB_1 ne fait intervenir qu'une seule collection, QDB_1 est déjà atomique.

let $t_1 :=$ **for** $\$h$ **in** $Collection(" * ")/hotel$ **where** $\$h/categorie = "3"$ **return** $(\$h/nom, \$h//client)$	QA_1^1
	$QDep^1$

La requête QDB_2 comporte trois ensembles de collection dont une générée par la QDB_1.

for $\$t$ **in** $\$t_1$ **return** $(\$t//client)$	QA_1^2
for $\$p$ **in** $Collection(" * ")/personne$ **return** $(\$p/prenom, \$p/nom)$	QA_2^2
for $\$l$ **in** $Collection(" * ")/livre$ **where** $contains(\$l/titre, "UNIX")$ **return** $(\$l/nom, \$l/titre)$	QA_3^2
$\$p/nom = \$l/auteur/nom$ **and** $\$p/nom = \$t//client$	$QDep^2$

3.7.3 Identification des sources

Une fois atomisée, chaque requête est ensuite analysée afin de repérer les sources intervenant dans la requête. On utilise l'accès au cache de descriptions de métadonnées pour étendre les chemins, ainsi un prédicat $/conference//auteur/nom$ (c'est-à-dire : un chemin dont la première étiquette est $conference$ que les dernières étiquettes sont $auteur/nom$ mais dont les intermédiaires peuvent être de n'importe quelle longueur et inclure diverses sortes d'étiquettes), pourra par exemple être étendu après consultation du cache de descriptions de métadonnées à : $/conference/invites/auteur/nom$ et $/conference/articles/auteurs/auteur/nom$.

```
fonction trouverSource ( requête_atomique)
 nom_collection <- recupererNomDeLaCollection ( requête_atomique)
 tableau_tous_chemin [] <- résoudreTousLesCheminsAbsolusPossibles (
                                                    requête_atomique)
 modifierRequête (tableau_tous_chemin) ;
 tableau_sources [] <- chercherSourcesParNomDeCollection (index_de_collection,
                                                          nom_collection)

  pour chaque  source_i  de tableau_sources
    vérifier que tableau_tous_chemin est inclu dans les chemins de  source_i
    ajouterDans (tableau_sources_valide,  source_i)
  fin pour
fin_fonction
```

Requête atomique	ensemble de chemins résolu	sources
QA_1^1	$Collection("HOTELS")/hotel/categorie$ $Collection("HOTELS")/hotel/nom$ $Collection("HOTELS")/hotel/clientele/client$	M3
QA_1^2	$Collection("HOTELS")/hotel/categorie$ $Collection("HOTELS")/hotel/nom$ $Collection("HOTELS")/hotel/clientele/client$	M3
QA_2^2	$Collection("PERSONNES")/personne/nom$ $Collection("PERSONNES")/personne/prenom$	W3
QA_3^2	$Collection("LIVRES")/livre/titre$ $Collection("LIVRES")/livre/auteur/nom$	W3, M3

Il se peut que la même source contienne les deux types de collections (par exemple, la source W3 contient à la fois la collection `LIVRES` et aussi la collection `PERSONNES`), ou qu'un type de document soit présent dans plusieurs sources à la fois (par exemple la collection `LIVRES` se retrouve dans les sources W3 et M3).

3.7.4 Création du plan d'exécution

Un plan d'exécution est une structure arborescente comportant au niveau des feuilles les sources à interroger et au niveau des nœuds, des opérateurs bien connus du monde relationnel : jointure, restriction, projection, etc. ainsi que des opérateurs de fonctions. L'arbre d'exécution de la requête exemple est donné à la figure 3.9.

Pour chaque requête atomique QA_i s'appliquant à une source physique, un opérateur (nommé *XSource*) est créé. Son rôle consiste à interroger directement l'adaptateur gérant cette source avec la requête atomique associée. Les requêtes de liaison $QDep$ permettent de construire les opérateurs de jointure (*XJoin*). La collection "LIVRES" se trouvant à la fois dans la source W3 et la source M3, un opérateur d'union (*XUnion*) est créé pour ces deux sources. Enfin, la requête $QMem$ de reconstruction est utilisée pour construire l'opérateur de reconstruction. Nous décrirons dans le chapitre 4 sur l'algèbre le fonctionnement détaillé de chacun des opérateurs.

3.7.5 Optimisation du plan d'exécution

À l'issu des étapes précédentes, nous obtenons un arbre algébrique permettant de définir le plan d'exécution. Ce plan d'exécution peut-être optimisé par des règles de réécriture d'expressions d'opérateurs. Des règles simples et classiques comme la remontée de restrictions et des projections peuvent être appliquées. Dans le cadre d'une architecture de médiation, une optimisation à base d'ordonnancement de jointures suivant les

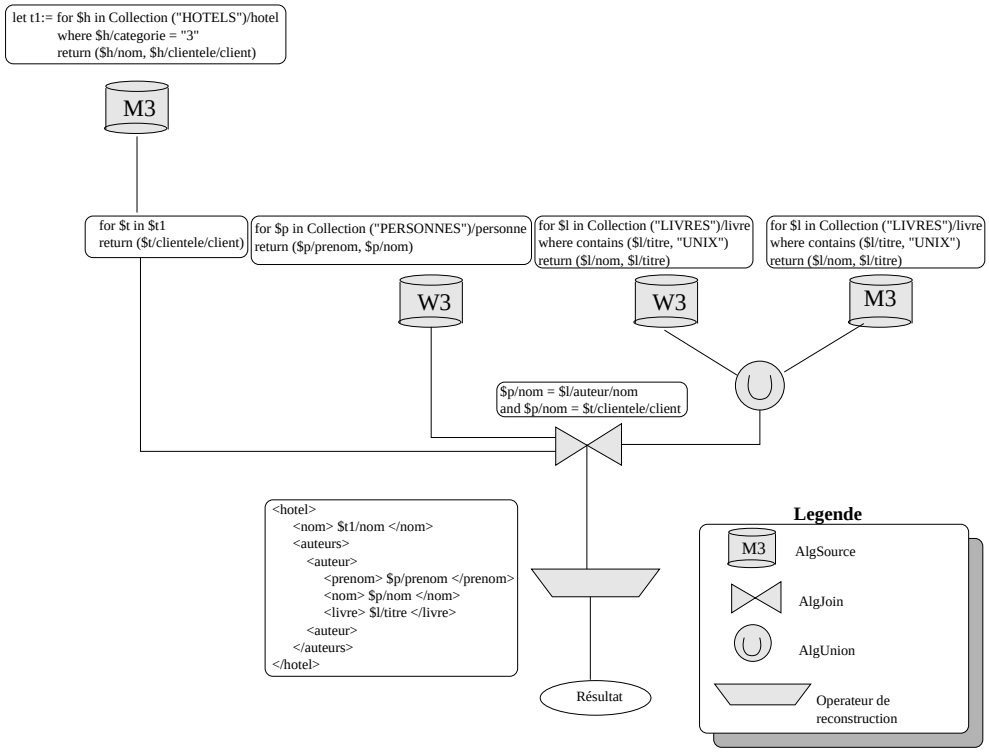


FIG. 3.9 – Plan d'exécution de la requête exemple

statistiques sur les collections, ainsi que des regroupement de sous-requêtes portant sur la même source peuvent être ajoutée. Toutes ces techniques d'optimisation manipulant un arbre algébrique sont possibles.

Nous définirons dans le chapitre suivant, le modèle de données, l'algèbre et les règles d'équivalence utilisées pour l'optimisation.

3.7.6 Recomposition

Le *recomposeur* transforme la structure interne des résultats en arbre en suivant la définition de construction donnée dans la requête. Pour cela, le recomposeur remplace les variables de la $QMem$ par les valeurs trouvées au fur et à mesure de l'évaluation. Un mécanisme de balisage créé la structure finale au fur et à mesure et insère les valeurs des variables résolues. Il se fait suivant l'algorithme suivant :

```
pour chacun des résultats
    lire structure de recomposition
    si variable
        chercher dans résultat la valeur de la variable
        générer valeur par SAX
    sinon
        générer balises par SAX
    fin si
fin pour
```

Le résultat construit pour la requête Q-1 est alors :

```
<results>
    <hotel>
        <nom>Hotel Marignan</nom>
        <auteur>
            <prenom>J.M.</prenom>
            <nom>Bach</nom>
            <titre>The design of the UNIX Operating System</titre>
        </auteur>

        <auteur>
            <prenom>Richard</prenom>
            <nom>Stevens</nom>
            <titre>UNIX Network Programming</titre>
        </auteur>
    </hotel>

    <hotel>
        <nom>Hotel du depart</nom>
        <auteur>
            <prenom>Jean-Marie</prenom>
            <nom>Rifflet</nom>
            <titre>La programmation sous UNIX</titre>
        </auteur>
        <auteur>
            <prenom>Jean-Michel</prenom>
            <nom>Moreno</nom>
            <titre>UNIX administration</titre>
        </auteur>
        <auteur>
            <prenom>Richard</prenom>
            <nom>Stevens</nom>
            <titre>UNIX Network Programming</titre>
        </auteur>
    </hotel>
</results>
```

3.8 Conclusion

Nous avons présenté dans ce chapitre l'architecture de médiation que nous avons utilisée dans l'ensemble des projets MIROWEB, XML-KM et MUSE. La version finale fait partie de la suite d'intégration *e-XMLMedia Suite*. Nous avons détaillé les différents

composants de cette architecture de médiation basée sur celle du DARPA I3.

Le nombre croissant et la disparité - en particulier les données semi-structurées- des sources de données dans les systèmes distribués hétérogènes posent des nouveaux problèmes pour le développeur d'adaptateurs, l'administrateur des bases de données, le développeur d'applications et l'utilisateur final. Le développeur d'adaptateur doit sans cesse faire face à l'émergence de nouveaux types de sources, et l'universalité du langage pivot est crucial dans son travail. Unifier de plus les moyens d'accès aux sources et aux adaptateurs simplifient considérablement son travail. L'administrateur des bases de données doit pouvoir participer au système d'intégration en offrant des nouveaux accès bases de données existantes sans remettre en cause les applications y accédant et les habitudes des utilisateurs. Le développeur d'application doit en plus des programmes existants, savoir d'une part comment intégrer des données dont les structures peuvent être complexes ou connues partiellement ; et d'autre part comment utiliser des données pouvant provenir de sources très diverses. La définition de standard d'interrogation et de communication est pour lui primordial. Il doit en plus intégrer un nouveau langage d'interrogation permettant de formuler des requêtes sur des données semi-structurées. Enfin pour l'utilisateur final, si les différentes sources peuvent lui être occultées, il faut qu'il puisse naviguer dans les données dans un style proche des hyperliens du web pour découvrir la structure des données au fur et à mesure.

Notre médiateur propose des solutions nouvelles pour résoudre en partie les problèmes de chacune de ces personnes.

À l'instar de ODBC/JDBC comme API standardisé pour interroger des applications sur des données relationnelles, rien n'a encore été standardisé pour l'interrogation d'applications sur des données semi-structurées. L'API XML/DBC bien que non standardisé est suffisamment riche et cohérent pour répondre à une telle demande. XML/DBC a été proposé par e-XMLMedia au W3C. L'utilisation de l'API XML/DBC au niveau des sources, des applications et des médiateurs permet de réaliser des architectures multi-niveaux. Un médiateur peut en effet apparaître comme une source de données pour un autre.

Le développement d'adaptateurs est rendu générique et répond aux standards en vigueur tant au point de vue des communications (SOAP, RMI), du langage d'interrogation (XQuery), de description des métadonnées (XML-Schema) et des transitions des données (SAX). L'utilisation des XML-Schema et de collections permet de décrire le contenu des sources et les exporter de façon portable. L'utilisation de XQuery est utilisé comme langage de requête, et de XML comme modèle de données à tous les niveaux permet une modularité des composants. XQuery tout comme XML étant standardisés par le W3C. Et enfin, l'utilisation des flux SAX entre les composants et les opérateurs permet de réaliser des opérations en série ou en parallèle suivant les propriétés bloquantes des opérateurs.

Pour la communication d'informations spécifiques à une architecture distribuée, nous avons utilisé aussi des formats basés sur XML. Le langage de coût est entièrement en XML et utilise pour les définitions des formules le standard MathML. Ce standard est utilisable

par beaucoup de logiciels mathématiques (MathLab, Maple, Mathematica, etc.) ce qui pourrait à terme permettre une connexion vers de tels logiciels pour les évaluations de coûts et des statistiques. Enfin le langage permettant aux adaptateurs de communiquer leur capacité est aussi représenté sous format XML. Cette approche consistant à baser tous les échanges de données en XML achève de décrire une intégration dans un contexte « tout-XML ».

Chapitre 4

Une méthode d'évaluation pour une algèbre semi-structurée

4.1 Introduction

Plusieurs algèbres ont été définies pour permettre l'interrogation de données semi-structurées. Certaines se basent sur une structure de graphe (algèbre IBM), d'autres s'appuient sur une structure de type tabulaire (algèbre Y_AT). Des systèmes comme XPeranto, AGORA ou Nimble transforme les requêtes XQuery en requête relationnelle et les résultats sous forme de tuple en arbre XML.

Un thème principal [Grahne et Lakshmanan 2000] émergeant de tous les langages de requête basés sur le semi-structuré est que la navigation est un composant essentiel et à part entière des requêtes. De plus, dû au manque de schéma rigide dans les données semi-structurées, la navigation amène plusieurs avantages dont le fait de pouvoir rapatrier des données indépendamment de la profondeur à laquelle elles se situent dans l'arbre. Ceci peut être réalisé grâce à des primitives de programmation comme les expressions rationnelles de chemins et les jokers ou caractères génériques (*wildcards*).

Nous décrirons une approche mixte basée sur des opérateurs relationnels étendus et des modèles de données sous forme de graphe utilisant des pointeurs indexés permettant d'optimiser le processus de reconstruction et de parcours de chemin.

4.2 Plan du chapitre

Nous exposerons tout d'abord l'état de l'art sur l'algèbre de données semi-structurées (section 4.3). Nous décrirons le processus d'évaluation que nous avons mis en œuvre dans la section 4.4. Dans la section 4.5 nous formaliserons les termes algébriques que nous employerons dans la suite. Dans la section 4.6 nous décrirons une algèbre physique permettant d'évaluer efficacement des requêtes sur XML. Pour cela, nous détaillerons dans 4.6.1 le modèle de données que nous avons utilisé pour la représentation des données internes et l'évaluation des requêtes sur ces données. Nous introduirons le concept de XTuple permettant de réaliser une évaluation de requête efficace sur des données XML. Ensuite les opérateurs seront dépeints dans la section 4.6.2, et nous montrerons comment ils s'appliquent aux XTuples que l'on aura introduit dans la section précédente. Nous donnerons sur ces opérateurs quelques règles d'équivalence permettant de simplifier certaines évaluations dans la section 4.7. Et enfin, nous conclurons dans la partie 4.8 du chapitre.

4.3 État de l'art

Le W3C (*World Wide Web Consortium*) a posé trois conditions qu'un mécanisme de manipulation de données XML doit satisfaire :

- *la recherche de chemin* : étant donné un chemin XML et une collection de données XML, le mécanisme devrait être capable d'identifier les positions dans lesquelles ce chemin apparaît ;
- *le filtrage* : une fois les chemins identifiés, ils doivent pouvoir être sélectionnés suivant les valeurs de leurs attributs spécifiques ;
- *la construction XML* : afin de supporter la transformation de documents, tout mécanisme de manipulation de données XML doit fournir une méthode de création de données à partir de données existantes.

En vue de satisfaire ces exigences, des langages de requêtes ont été proposés, et des algèbres ont été conçues. C'est ce que nous allons développer dans cette section.

4.3.1 Algèbre

Les représentations algébriques de requêtes ont été largement étudiées depuis la définition de Codd du modèle relationnel [Codd 1972]. Beaucoup de travaux ont aussi été réalisés dans l'algèbre des bases de données orientées objets [Fegaras et Maier 1995].

Avec l'apparition des données semi-structurées avec LORE, des travaux sur une algèbre adaptée à ce type de données ont vu le jour. Puis lors de l'évolution des données

semi-structurées vers XML, il a clairement été établi qu'il fallait définir une algèbre pour XML.

Les exigences demandées pour une algèbre pour XML sont :

- *la puissance d'interrogation* : l'algèbre doit permettre l'évaluation de langage de requête en respectant les spécificités de XML. En particulier, elle doit pouvoir supporter des primitives de navigation, plusieurs sortes de variables (valeurs atomiques, sous-arbres, étiquette) ainsi que des fonctions de Skolem et des recherches de chemins complexes ;
- *un support pour un typage flexible* : XML permet la flexibilité du typage, et beaucoup de langages de requête XML ne sont pas typés. Mais on doit néanmoins pouvoir bénéficier des propriétés des langages structurés et des schémas éventuels. C'est à dire que l'algèbre doit pouvoir à la fois supporter des types flexibles mais aussi pouvoir gérer des structures plus rigides que sont les schémas ;
- *un support pour l'optimisation* : bien sûr, l'algèbre devrait pouvoir offrir des possibilités d'optimisation par un certain nombre d'équivalences.

Pour répondre à ces exigences, plusieurs algèbres pour XML (IBM [Beech *et al.* 1999], YAT [Christophides *et al.* 2000], AT&T[Fernandez *et al.* 2001], LORE [McHugh et Widom 1999b]) ont été proposées, et seule l'algèbre proposée par l'AT&T a été retenue et publiée dans le papier de travail [Consortium 2000] en décembre 2000. D'autres algèbres comme NIAGARA [Galanis *et al.* 2001] et TAX [Jagadish *et al.* 2001] ont fait leur apparition par la suite.

4.3.1.1 Algèbre d'IBM et Niagara

L'algèbre d'IBM [Beech *et al.* 1999] a été proposée en septembre 1999 par les chercheurs d'IBM, Oracle et Microsoft. Elle propose des *opérateurs algébrique* opérant au dessus d'un modèle de données basé sur une structure de graphe. Le modèle présenté est un modèle purement logique et rien n'est précisé quand à la représentation physique des opérateurs. Outre les opérateurs standards de requête, l'algèbre d'IBM comporte des opérateurs de restructuration permettant de créer des nouveaux documents XML à partir de fragments de document sélectionnés.

L'opérateur de navigation est :

$$\phi[type\_arc, nom](collection\_noeuds)$$

type_arc représente un type d'arc qui peut être E (Element), A (Attribut) ou R (Référencement). *collection_noeuds* est l'ensemble des nœuds du graphe qui sont origines de l'arc de type *type_arc* et de nom *nom*. On abrégera $\phi[E, b](a)$ en a/b, $\phi[A, b](a)$ en $a/@b$ et $\phi[R, b](a)$ en $a/b >$.

```
<?xml version=1.0" encoding="ISO-8859-1" standalone="yes"?>
<liste>
    <personne>
        <nom>Cover</nom>
        <age>36</age>
        <prenom>Harry</prenom>
        <adresse>
            <rue>
                <numero>11</numero>
                <numero>Maupassant</numero>
            </rue>
            <ville>Paris</ville>
        </adresse>
        <profession>administrateur système</profession>
    </personne>
    <personne>
        <nom>Well</nom>
        <age>26</age>
        <prenom>Rose</prenom>
        <adresse>
            <rue>
                <numero>5</numero>
                <numero>rue du marché</numero>
            </rue>
            <ville>Versailles</ville>
        </adresse>
        <profession>ingénieur développement</profession>
    </personne>
</liste>
```

Doc. 4.1: Document XML exemple

Par exemple, en utilisant le document XML présenté dans le document 4.1, la requête Q4 suivante :
Q4 : *Trouver toutes les personnes dont l'âge est inférieur à 30 ans*

se traduirait dans cette algèbre par :

Q4 :

$$\sigma[value(x/personne/age/\ data) < 30](x : child(/liste))$$

Où σ est l'opérateur de sélection et *child* est un champ d'un arc retournant le nœud référencé par cet arc. Tous les opérateurs travaillent sur des collections d'arcs ou de nœuds. La jointure $\otimes$ prend deux collections de nœuds comme arguments. Les opérateurs d'exposant ϵ et de retour ρ sont définis pour les requêtes voulant présenter des composants ou des fragments de documents. D'autres opérateurs sont aussi proposés comme l'ordonnancement des nœuds Σ, ou μ permettant d'appliquer une fonction spécifique à une collection d'arcs ou de nœuds. Des opérateurs comme la fermeture de Kleene (*Kleene star*) $*$ sont aussi définis.

Les principaux inconvénients de l'algèbre d'IBM sont :
- au moment de l'écriture de la requête, le type (élément, attribut, référence) de chaque arc doit être connu ;
- l'algèbre définit beaucoup d'opérateurs et préserve explicitement l'affectation de toutes les variables apparaissant dans la requête en les incluant dans les requêtes intermédiaires, ce qui conduit à des structures complexes lors de la représentation d'une requête ;
- peu de règles d'optimisation ont été définies.

C'est pour contrer ces inconvénients que Niagara propose de ne pas travailler sur des ensembles de nœuds mais sur des *sacs* de nœuds. Un sac de nœuds contient non seulement le nœud sélectionné mais tous les nœuds intermédiaires qui ont été visités le long du chemin pour arriver jusqu'à ce nœud. La requête exemple Q4 précédente devient dans cette algèbre :

$$\epsilon(*.personne)[\sigma_{*.personne.age<30}[\phi(*.personne)[s(liste.xml)]]]$$

s est l'opérateur utilisé pour spécifier la source d'où proviennent les données, ϕ est l'opérateur permettant de suivre un chemin dans les éléments d'un groupement, σ est l'opérateur de sélection et ϵ présente tous les chemins spécifiés. Les autres opérateurs définis dans cette algèbre sont les opérateurs classiques de l'algèbre relationnelle : l'union $\cup$, la différence $\setminus$, le produit cartésien $\times$ et la jointure $\bowtie$.

Des optimisations et des règles de réécritures ont été aussi présentées.

4.3.1.2 Algèbre TAX

L'algèbre TAX [Paparizos *et al.* 2002] [Jagadish *et al.* 2001] étend l'algèbre relationnelle en considérant comme unité de base de manipulation, des collections d'arbres étiquetés ordonnés au lieu de relations.

TAX introduit la notion de *pedigree* qui est un identifiant « léger » pour chaque élément des arbres. Ce *pedigree* est un attribut « invisible » supplémentaire dans chacun des éléments de l'arbre traité, qui permet de grouper, ordonner et supprimer les duplicata. Un *pedigree* est le couple composé de l'identifiant du document et de la position de l'élément dans ce document.

En dépit de la complexité potentielle due à la structure d'arbre et à l'hétérogénéité d'une collection, TAX n'a que quelques opérateurs de plus que l'algèbre relationnelle.

4.3.1.3 Algèbre Y_AT

L'algèbre Y_AT [Christophides *et al.* 2000] a été développée pour Y_AT, un système d'intégration basé sur XML.

L'algèbre Y_AT est caractérisée par deux opérateurs nommé $Bind$ et $Tree$. À partir d'une structure arborescente arbitraire XML, $Bind$ extrait les informations utiles à l'évaluation, et produit une structure nommée Tab comparable à une relation $\neg 1NF$. Sur ces structures Tab, il est ensuite possible d'appliquer les opérateurs relationnels standards (jointure, sélection, projection, union, différence, etc.) L'opérateur $Tree$ correspond à l'opération l'inverse de $Bind$ puisqu'il permet de transformer la structure Tab en une structure arborescente XML.

La requête Q4 dans cette algèbre est exprimée par la figure 4.1

L'algèbre Y_AT transforme une structure XML en une structure tabulaire. De cette manière, elle peut manipuler des opérateurs standards de l'algèbre relationnelle (jointure, sélection, projection). Une opération de construction de l'arbre est ensuite mise en œuvre.

4.3.1.4 Algèbre LORE

Le système LORE [McHugh et Widom 1999b] développé à Stanford est un SGBD complet conçu spécifiquement pour des données semi-structurées. Dans ce système, une requête est transformée en un plan d'exécution utilisant des opérateurs algébriques ($Select$, $Project$, etc.) L'objectif de ce système étant d'optimiser au maximum son modèle de coût. Ce plan d'exécution doit pouvoir générer plusieurs autres plans d'exécution équivalents

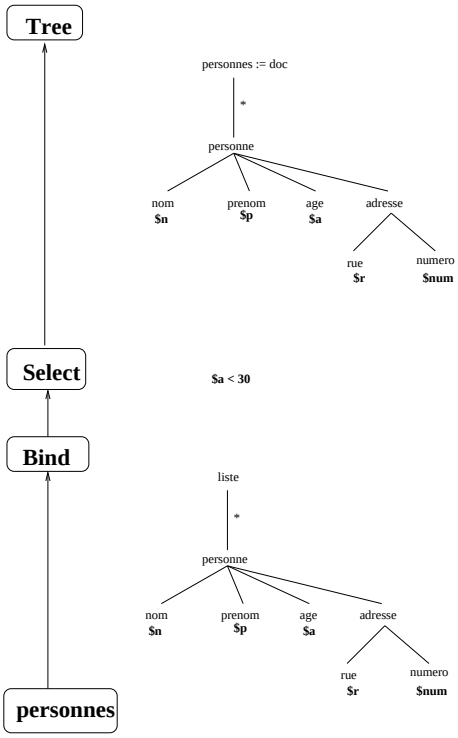


FIG. 4.1 – Requête Q4 sous forme algébrique

afin de choisir le plan le plus optimal. Pour cela, deux opérateurs supplémentaires [McHugh et Widom 1999a] *Glue* et *Chain* appelés *opérateurs de rotation* ont été implémentés. Leur but est de préciser que les deux sous-arbres algébriques peuvent être intervertis à ce point précis de rotation. Le plan d'exécution comportant ces opérateurs de rotation est appelé un *plan logique*. Les multiples plans obtenus par les différentes combinaisons en utilisant les points de rotations sont appelés *plans physiques*. Le plan logique correspondant à la requête Q4 est donné à la figure 4.2 en (a), et deux plan physiques possibles parmi toutes les combinaisons sont donnés en (b) et (c).

L'algèbre LORE reste très liée au modèle OEM et aux structures de données et d'indexation associés.

4.3.1.5 Algèbre AT&T

L'algèbre d'AT&T [Fernandez *et al.* 2001] est directement inspirée par SQL, OQL et l'algèbre relationnelle imbriquée. Les opérateurs de cette algèbre sont basés sur des opérateurs d'itération. La requête Q4 devient alors :

Q4 :

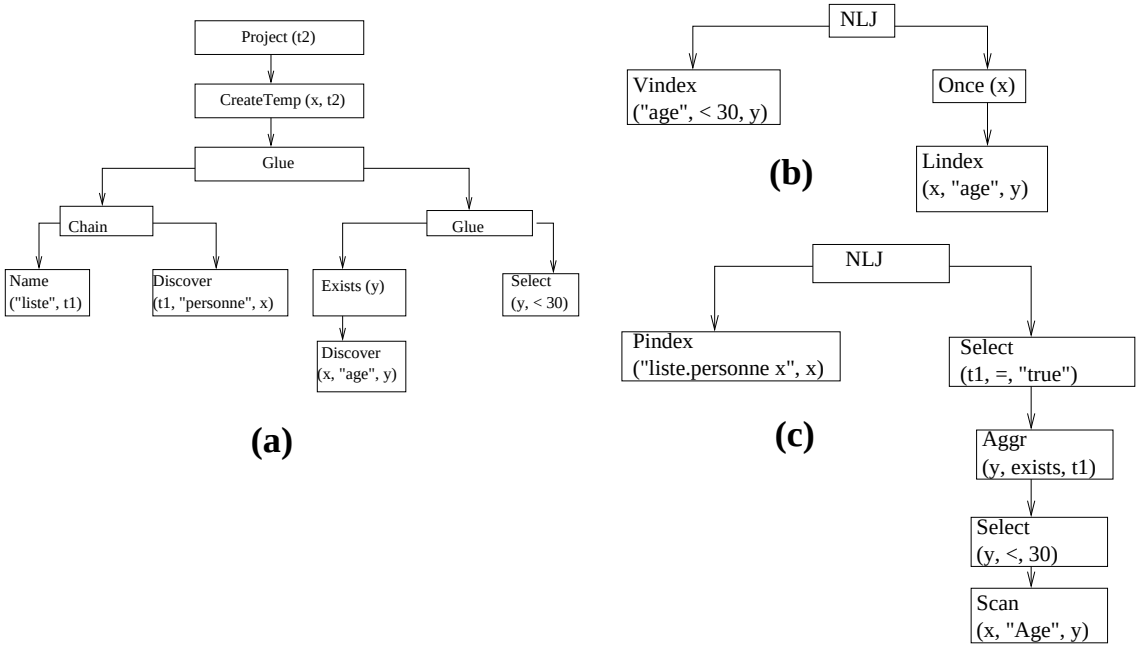


FIG. 4.2 – Plans logique et physiques de Q4

```
for  pers  in liste/personne
do
     where  pers/age/data() < 30
         do
         pers
```

L'opérateur d'itération est une expression for qui parcourt les éléments d'un document suivant une expression de chemin. Une opération de sélection est réalisée grace à une clause $where$ permettant d'exprimer les critères de restrictions. Une opération de jointure se fait par l'imbrication de deux boucles for. L'opérateur de chemin / se fait aussi par l'utilisation d'un boucle for. Ainsi $liste/personne$ se résout par la fonction suivante :

```
for  fils  in children (liste)
do
    match  fils
        case  x : personne [type_personne]  do  x
        else ()
```

Un ensemble de règles de simplification des for inutiles par ré-ordonnancement ont été décrites [Fernández et Siméon 2000]. L'avantage de cette algèbre est qu'étant très proche du langage XQuery, très peu de conversion est à faire pour le passage de l'un à l'autre. Les opérateurs sont fortement typés. L'optimisation se fait au niveau des boucles (comment regrouper les boucles, comment les échanger). C'est cette algèbre qui a été retenue par le W3C [Consortium 2000].

4.3.2 Synthèse

Pour évaluer des requêtes sur des données semi-structurées, il faut définir un langage adapté à ce nouveau type de donnée. Ensuite, il faut proposer une algèbre permettant de modéliser de façon formelle une requête portant sur des données semi-structurées. Cette algèbre doit être suffisamment puissante et complète pour pouvoir permettre d'exploiter toutes les spécificités des données semi-structurées, être assez souple pour pouvoir accepter un typage très lâche tout en ayant des primitives de navigation assez puissantes pour pouvoir parcourir les structures et utiliser des schémas rigides lorsque le besoin se fait sentir. Enfin, afin d'être efficace, cette algèbre doit disposer de suffisamment de règles d'équivalence pour permettre différentes façons d'exécution possibles pour une même requête.

À part l'algèbre de LORE développée pour le système LORE avec des opérateurs logiques développés en tenant compte des opérateurs physiques spécifiques à ce système, les autres algèbres (AT&T, IBM, Niagara, TAX) sont des algèbres autonomes. L'algèbre Y_AT a été développée plus spécifiquement dans le cas de systèmes d'intégration basés sur XML, et les stratégies d'optimisation se concentrent sur l'interrogation efficace de requêtes distribuées. Niagara utilise des opérateurs basés sur des collections. L'algèbre AT&T, choisie comme standard du W3C, se base sur des expressions de requête sur un langage de haut niveau.

4.4 Processus d'évaluation

La figure figure 4.3 montre le processus complet de l'évaluation d'une requête. Le composant ① effectue la normalisation d'une requête XQuery en une forme plus « canonique » (désimbrication des boucles imbriquées). La partie reconstruction (*return*) est conservée à part, et seuls les résultats « mis à plat » seront retournés. L'identification des sources s'effectue dans le composant ② et consiste à décomposer une requête en sous-requêtes effectuables par les adaptateurs associés. La réponse à la requête est passée à l'adaptateur dans un composant ③ de transformation d'un document arborescent de type XML/SAX en une structure plus propice à nos opérations que nous appellerons XTuple. La composition des résultats et l'évaluation se fait ensuite dans l'arbre algébrique d'exécution en ④ enfin la structure XTuple résultat est transformée en document XML en ⑤ puis reconstruite suivant la forme demandée à la clause *return* de la requête initiale en ⑥.

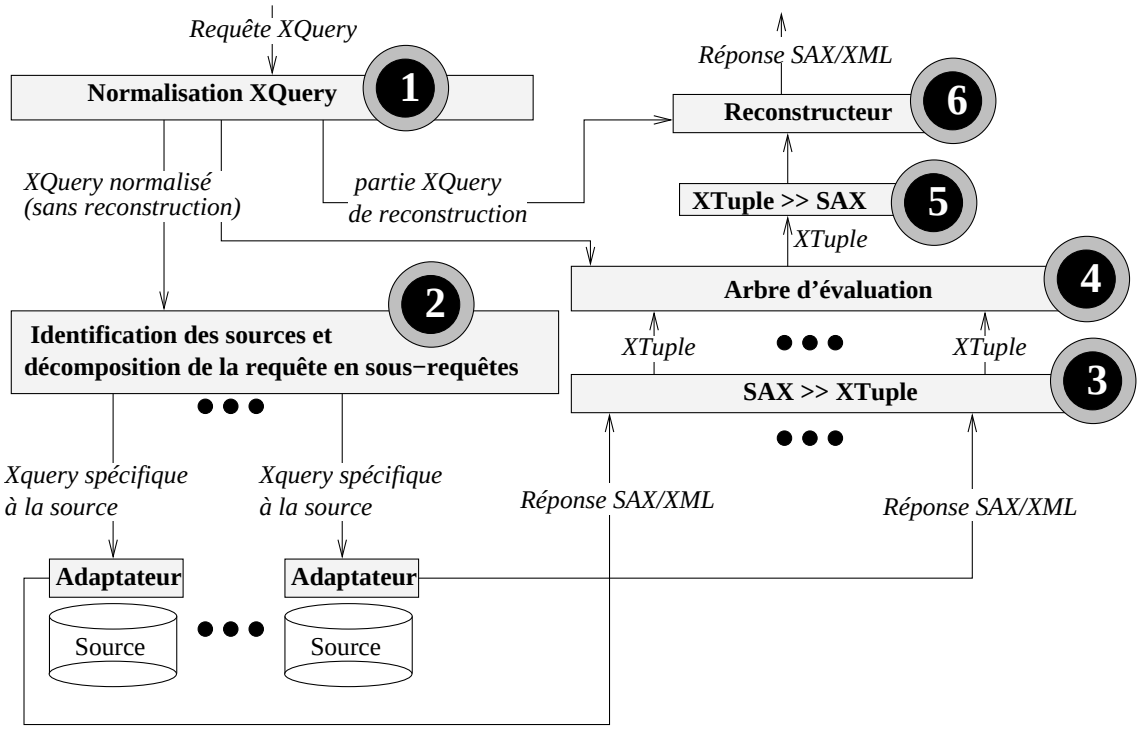


FIG. 4.3 – Processus d'évaluation d'une requête

4.5 Modèle de données formel et algèbre pour XML

Afin de pouvoir décrire précisément l'algèbre utilisée, nous introduisons les notions et notations utilisées par la suite. Nous utiliserons les notations introduits par [Beech *et al.* 1999].

4.5.1 Formalisation

Un document XML est représenté par un *graphe orienté*. Ce graphe comporte des *nœuds* (ou sommet) et des *arcs*. Il y a deux types de nœuds : les nœuds représentant les *éléments* du document, et les nœuds représentant les *valeurs*. Il y a aussi trois types d'arcs : les *arcs d'éléments*, les *arcs d'attributs*, et les *arcs de référencement*. Les arcs d'éléments sont les arcs relatifs à la relation d'un nœud élément avec ses nœuds fils qui peuvent être soit des nœuds éléments, soit des nœuds valeur. Les arcs d'attributs sont les arcs reliant un élément à ses attributs. Les arcs de référencement sont utilisés pour les éléments référençant des éléments externes *via* des pointeurs (IDREF, XLink, URI). L'ordonnancement des nœuds fils d'un élément parent est aussi à considérer. Les nœuds référencés par des arcs d'éléments ou les arcs de référencement sont ordonnés suivant

l'ordre dans lequel ils apparaissent dans le document et les attributs n'ont pas d'ordre.

Définition 4.1 : Arbre

Soit $\mathcal{T}(r, \mathcal{S}, \mathcal{E}, \mathcal{A}, \mathcal{P})$ l'arbre représentant le modèle de donnée de documents XML. $\mathcal{S}$ est l'ensemble des nœuds du graphe, les nœuds peuvent être de type élément ou de type valeur ($\mathcal{S} = \mathcal{S}_{element} \cup \mathcal{S}_{int} \cup \mathcal{S}_{string} \cup ...$). $\mathcal{E}$ représente l'ensemble des arcs d'éléments, $\mathcal{A}$ l'ensemble des arcs d'attributs, $\mathcal{P}$ l'ensemble des arcs de référencement (Pointeurs). r est le nœud particulier ($r \in \mathcal{S}$) correspondant à la racine de l'arbre.

Chaque élément XML est représenté par un nœud élément $v \in \mathcal{S}_{element}$. v a un identifiant unique, logique, abstrait et non-modifiable et indépendant du support physique. Chaque valeur est représentée par un nœud valeur $v \in \mathcal{S}_{type(v)}$. Un nœud valeur n'a pas d'identifiant unique et possède un *type* (string, int, float, etc.) et une *valeur* (42, "42", 4.2, etc.).

Chaque arc $e \in \mathcal{E}$ est une relation de $parent \in \mathcal{S}_{element}$ vers $fils \in \mathcal{S}$ avec $nom \in T_{nom}$ où T_{nom} est l'ensemble des chaînes de caractères valides comme nom d'étiquette. Dans le cas où $fils$ est un nœud élément, le nom nom correspond à l'identifiant générique de $fils$. Dans les autres cas, nom vaut soit *data*, *comment* ou *pi* (resp. donnée, commentaire ou *processing instructions*) suivant le type.

Chaque arc $e \in \mathcal{A}$ est une relation de $parent \in \mathcal{S}_{element}$ vers $fils \in \mathcal{S}_{valeur}$ avec $nom \in T_{nom}$.

Chaque arc $e \in \mathcal{P}$ est une relation de $parent \in \mathcal{S}_{element}$ vers $fils \in \mathcal{S}$ référencé par une référence $refarc \in \mathcal{P}(\mathcal{E} \cup \mathcal{A})$

On illustrera la définition par le document XML d'exemple suivant dont le graphe est représenté par la figure 4.4 :

```
<societe nom="SuperOrdi">
  <magasin id="mag1" nom="Planete Clavier"
                     pageweb="http://www.planete-clavier.com"/>
  <magasin id="mag2" nom="Planete Ordi">
    <inventaire>
      <clavier quantite="36" touches="101" fournisseur="mag1">
      <ecran quantite="26" taille="42'" fournisseur="mag2" >
        <description>Un ecran <em>de tres haute</em>qualite</description>
    </inventaire>
  </restaurant>
</Societe>
```

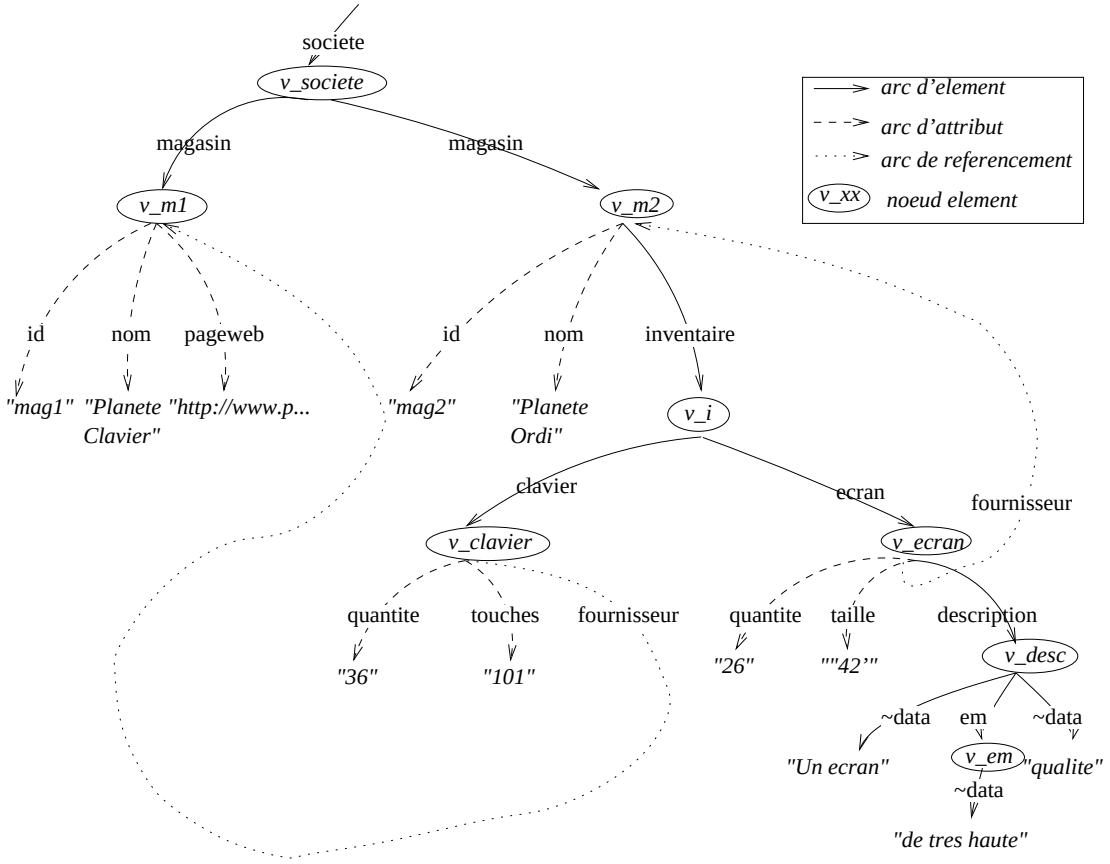


FIG. 4.4 – Arbre associé au document exemple

4.5.2 Navigation

La navigation dans un chemin à travers un graphe est définie par l'intermédiaire de l'opérateur ϕ de la manière suivante :

$$\phi[type\_arc, nom](coll\_noeud)$$

où

- $type\_arc \in \{\mathcal{A}, \mathcal{E}, \mathcal{P}\}$
- $nom \in T_{nom} \cup regexp$ où $regexp$ est une expression rationnelle, par exemple $(a|A)d[d]ress.*$ décrit tous les noms commençant par $adress$, $Adress$, $Address$ ou $address$. . est le caractère générique signifiant n'importe quel caractère, et $*$ exprime que le motif précédent est répété aucune, une ou plusieurs fois.
- $coll\_noeud \subset \mathcal{S}$

On utilisera pour raison de simplicité la syntaxe abrégée de XPath :

- a/b pour $\phi[\mathcal{E}, b](a)$
- $a/@b$ pour $\phi[\mathcal{A}, b](a)$
- $a/>b$ pour $\phi[\mathcal{P}, b](a)$

On distingue également des *axes*, qui sont des propriétés implicites des arcs, on peut citer entre autres, l'axe $fils$ d'un arc qui retourne le nœud destination de l'arc, ou l'axe *parent* retournant la source de l'arc. L'opération $fils$ est souvent utilisée avec l'opération de navigation, par exemple :

$$fils(\phi[\mathcal{E}, b](fils(\phi[\mathcal{E}, a](racine))))$$

signifie : à partir du nœud racine, prendre l'arc de nom a, puis prendre le nœud fils donc, le nœud destination de cet arc, rechercher un arc nommé b, et enfin prendre le nœud destination de cet arc. En XPath, cela s'écrirait sous la forme : $racine/a/b$.

Fermeture de Kleene La fermeture de Kleene (*Kleene Star*) notée $*$, permet de répéter une opération autant de fois que nécessaire. La syntaxe

$$[f(x), n](x : expression) \Leftrightarrow \underbrace{f \circ f \circ ... \circ f}_{n\ fois}(x) \Leftrightarrow \underbrace{f(f...f(x)))}_{nfois}$$

permet de dire que la fonction $f(x)$ est appliquée n fois.

on notera $[f(x)](x : expression)$ si la fonction peut-être composée un nombre quelconque de fois (*ie.* n peut varier de 0 à ∞) Ainsi

$$fils(\phi[\mathcal{E}, b](*fils(\phi[\mathcal{E}, \#](fils(\phi[\mathcal{E}, a](racine))))))$$

veut dire A suivi de autant fils possibles, suivi d'un B. Ce qui abrégerait dans la syntaxe XPath par $A//B$. # désignant n'importe quel nom.

Définition 4.2 : Arc adjacent

Un arc est dit *adjacent* à un autre si le nœud destination de l'un est le nœud source de l'autre.

Définition 4.3 : Chemin

On appellera *chemin* $\boldsymbol{C}$ une séquence ordonnée d'arcs adjacents allant d'un nœud à un autre. Le chemin d'un nœud à un autre n'est pas obligatoirement unique (on peut avoir plusieurs arcs arrivant à un nœud - cas des arcs de référencement). Un chemin sera noté par la suite soit en utilisant la notation à base d'opérateurs de navigation, soit la syntaxe abrégée XPath.

Définition 4.4 : Sous-chemin

Un chemin $\boldsymbol{C}_1$ est un *sous-chemin* d'un chemin $\boldsymbol{C}_2$ si la séquence des arcs associée à $\boldsymbol{C}_1$ est inclus dans la séquence de arcs associée à $\boldsymbol{C}_2$. Un $\boldsymbol{C}_1$ est appelé *préfixe* de $\boldsymbol{C}_2$, si $\boldsymbol{C}_1$ est un sous-chemin de $\boldsymbol{C}_2$ et si le premier arc de $\boldsymbol{C}_1$ coïncide avec le premier arc de $\boldsymbol{C}_2$. On notera $\boldsymbol{C}_1 \prec \boldsymbol{C}_2$ le fait que $\boldsymbol{C}_1$ soit un préfixe de $\boldsymbol{C}_2$.
On notera $\boldsymbol{C}_1 \sqcap \boldsymbol{C}_2$ un préfixe commun des expressions de chemins $\boldsymbol{C}_1$ et $\boldsymbol{C}_2$. et $\boldsymbol{C}_1 \dot{\sqcap} \boldsymbol{C}_2$ le plus grand préfixe commun des expressions de chemins $\boldsymbol{C}_1$ et $\boldsymbol{C}_2$. Enfin on désignera par $\perp$, l'expression de chemin nul.

4.6 Une algèbre physique pour XML : XAlgèbre

En algèbre relationnelle les requêtes et les évaluations de requêtes sont axées sur la notion de *tuple*. Un tuple est simplement un tableau de *valeurs simples* correspondant à des *attributs*. De sorte qu'un ensemble de tuples (ou relation) résultat d'un opérateur d'algèbre ne se résume que par un tableau à deux dimensions de valeurs. Les opérateurs de base de l'algèbre relationnelle sont l'union, la différence, la projection et la sélection. Avec l'algèbre relationnelle, chaque opérateur prend un ou plusieurs arguments de type relation et renvoie une relation. D'autres opérateurs (jointure, intersection) composés d'opérateurs de base sont venus enrichir cette algèbre. Des règles de simplification d'expressions permettent d'optimiser les requêtes. La simplicité de cette algèbre est à la base de l'efficacité de l'algèbre relationnelle. C'est partant de cette constatation que l'on est amené à se demander s'il n'est pas possible d'adapter les opérateurs et travaux déjà réalisés pour le traitement des requêtes XQuery.

La structure arborescente et hétérogène des données semi-structurées ne permet pas de pouvoir classer de façon simple un ensemble de résultats. Une première approche est de mettre les arbres résultats « à plat », et donc considérer chaque nœud comme un attribut. Le problème est qu'un arbre résultat n'a pas forcément les mêmes attributs d'un arbre à l'autre, et donc que les *domaines* d'attributs sont différents pour les opérateurs. De plus, les liens père-fils entre les nœuds sont perdus ou alors difficilement récupérables. Une deuxième approche est de conserver tout l'arbre résultat et de le considérer comme unité de base de la même manière que le tuple dans l'algèbre relationnelle. Cette approche nécessite de pouvoir accéder à un niveau quelconque de l'arbre. La manipulation d'un arbre pour les opérateurs est de loin plus complexe que la manipulation d'un tuple du fait de leur structure plus riche et de l'hétérogénéité de leurs données. Elle nécessite de plus l'emploi fréquent des opérateurs de navigation d'arbres ([Jagadish *et al.* 2001] [Paparizos *et al.* 2002] [Wu et Jagadish 2002]) Une opération fréquemment utilisée pour évaluer les

requêtes orientées-objet est la navigation par traversée d'identifiants d'objets [Gardarin *et al.* 1996a]. La navigation d'arbres est une procédure coûteuse puisqu'elle consiste à partir d'une racine, à explorer chacun des chemins possibles en les comparant successivement au chemin de référence.

4.6.1 Modèle de données de la XAlgèbre

L'idée est de se dire qu'au niveau du médiateur, c'est l'opération de manipulation d'arbres (et en particulier la navigation) qui s'avère très coûteuse mais qu'il y a un moment où elle est de toute façon inévitable : lorsque les adaptateurs répondent aux sous-requêtes du médiateur, il est nécessaire de construire l'arbre à partir du flux de communication. Le principe que nous proposons est d'analyser finement la requête à l'aide des métadonnées afin de déterminer tous les chemins qui seront utiles au niveau de chacun des opérateurs. Il est alors possible de référencer et manipuler tous les nœuds en fonction des chemins ainsi calculés lors de la récupération des résultats par le médiateur. Nous créons ainsi à la volée des structures (que nous nommerons par la suite *XTuple*) lors de la construction des arbres récupérés.

Définition 4.5 : Référence

À un nœud v, on peut associer une *référence* $\uparrow(v)$ sur ce nœud, permettant d'y accéder de manière directe (notion d'adressage et de pointeur) ; à partir d'un chemin donné, on obtient ainsi une ou plusieurs références sur des nœuds correspondant à ce chemin.
Soit $\mathcal{C}_T$ l'ensemble des chemins d'un arbre $T \in \mathcal{T}$ et soit $\boldsymbol{C} \in \mathcal{C}_\mathcal{T}$; on appellera $\boldsymbol{R}(\boldsymbol{C})$ la fonction qui à partir du chemin $\boldsymbol{C}$ renvoie l'ensemble des références $R = \{\uparrow(v_1), \uparrow(v_2), ..., \uparrow(v_k)\}$ associées à ce chemin dans l'arbre T.

Définition 4.6 : XTuple

On appellera *XTuple* X un couple $(\mathcal{R}_X, \mathcal{T}_X)$ constitué d'un ensemble de références $\mathcal{R}_X = \{\mathcal{R}_1 \cup ... \cup \mathcal{R}_n\}$ et d'un ensemble $\mathcal{T}_X = \{T_1 \cup ... \cup T_n\} \subset \mathcal{T}$ d'arbres référencés ; avec $\forall i \in [1..n], \mathcal{R}_i \subset \mathcal{C}_{T_i}$.

Construction de XTuple Les adaptateurs communiquent les résultats sous forme XML au médiateur. Celui-ci analyse le document XML à l'aide de flux SAX afin de construire l'arbre DOM associé (figure 4.5)

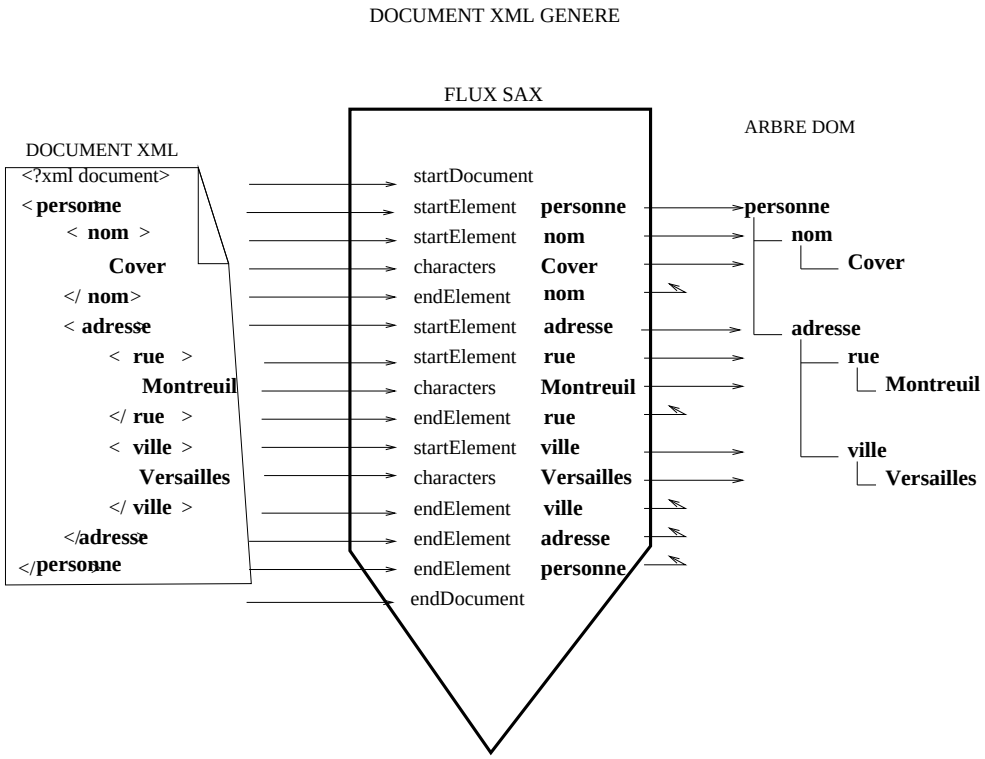


FIG. 4.5 – Génération d'un arbre DOM à partir d'un flux SAX

D'après la figure 4.5, on voit qu'il est peu coûteux d'indexer l'arbre *au moment même* de la transformation du flux SAX en arbre DOM. Les structures de manipulation de ces arbres résultats sont appelées *XTuple* et se décrivent ci-dessous.

Un *XTuple* est constitué des attributs *nécessaires* aux opérateurs correspondant à des *chemins* dans un arbre ainsi que d'une collection d'arbres (appelée aussi *forêt*). Les valeurs associées aux attributs sont en réalité des références vers les nœuds correspondants dans la forêt d'arbres. De cette façon, il est possible d'appliquer simplement les opérateurs relationnels sur les tuples, et de maintenir la description riche des arbres semi-structurés (liens père-fils, chemins complexes, etc.)

Par analogie avec l'algèbre relationnelle, une collection de XTuples est appelé *XRelation*.

La figure 4.6 représente une XRelation composée au moins de deux XTuples. La partie gauche du tableau donne les XAttributs, c'est-à-dire les références aux nœuds de la partie droite associées aux chemins XPath de sa colonne. À chaque colonne de chacun

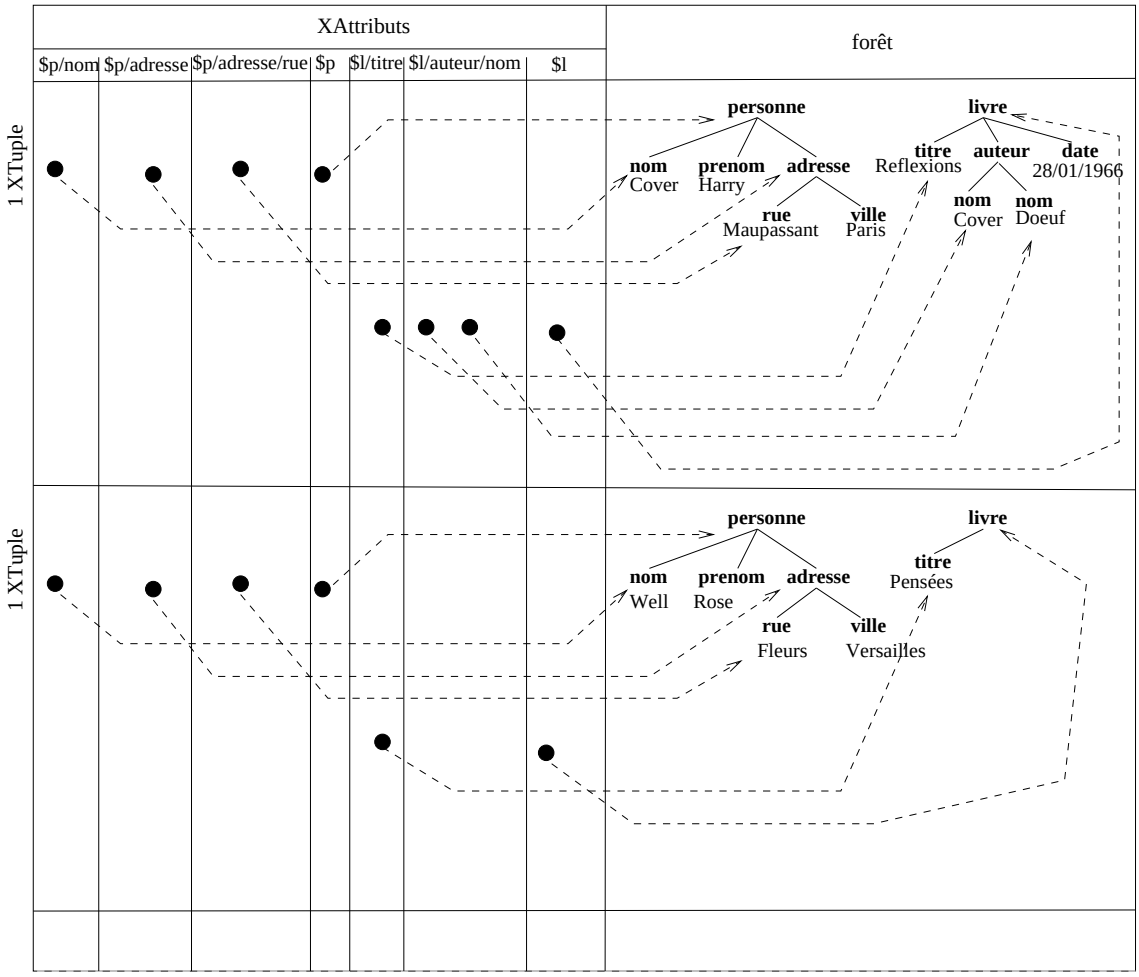


FIG. 4.6 – Structure d'un XTuple

des XTuples, il peut y avoir zéro, une ou plusieurs références suivant que le chemin dans les arbres associés est pas, une fois ou plusieurs fois représenté (attribut null, présent ou multivalué). La partie droite du tableau donne les instances d'arbres manipulées par le XTuple.

Trouver les Xattributs nécessaires à chaque opérateur se fait lors de la construction du plan d'exécution. En effet, il est possible de spécifier pour chaque chemin en fonction des besoins des autres opérateurs, les chemins qu'il est utile de conserver. Pour les nœuds multivalués, chaque Xattribut d'un tuple peut référencer plusieurs nœuds d'un arbre. Les chemins sont pré-calculés lors de l'analyse de la requête en fonction des prédicats demandés. Les chemins comportant des expressions génériques (joker) sont résolus.

4.6.2 Opérateurs

L'algèbre XAlgebre comporte à la fois des opérations relationnelles pour traiter les tables de XAttributs et à la fois des la navigation dans des arbres XML. Les documents XML sont envoyés au médiateur sous la forme de flux d'évènement (SAX dans l'implémentation). Les XTuples sont créés au fur et à mesure que les documents XML sont reçus depuis les adaptateurs. Les opérateurs non bloquants fonctionnent en « pipeline » sur le flux de XTuples. Les opérateurs bloquants ont besoin de l'instantiation complète de tous le flux d'entrée en mémoire. En général, les opérateurs N-aire non-bloquants parallélisent ses flux d'entrée.

La procédure d'évaluation de chaque opérateur est décomposée en deux phases : une phase d'initialisation et une phase d'exécution.

1. *La phase d'initialisation* : cette phase analyse le(s) XRelation(s) en entrée ainsi que les paramètres associés à l'opérateur afin de déterminer quelles seront les opérations exactes à réaliser quand les XTuples arriveront. Par exemple, pour une opération faisant fusionner des arbres, la phase d'initialisation consistera à déterminer à quelle référence de nœud le nouveau sous-arbre devra être lié et quels seront les chemins communs. De la sorte, la phase d'exécution sera efficace puisque la majeure partie du traitement aura déjà été effectuée.
2. *La phase d'exécution* : cette phase est réalisée lors de l'évaluation de la requête et commence lorsque les premiers XTuples arrivent en entrée. Le traitement des XTuples se fait en suivant les indications préparées par la phase d'initialisation.

4.6.2.1 Source ($\mathcal{S}$)

XSource est l'opérateur de départ permettant de traiter une source de données XML.

L'opérateur $XSource$ (noté $\mathcal{S}$) permet de soumettre une requête XQuery à une ou plusieurs collections dans une ou plusieurs sources. Une collection est identifiée par son nom. Une source est accédée *via* son adaptateur et est identifiée par son nom. L'opérateur $XSource$ renvoie en sortie le résultat de la requête effectuée par l'adaptateur sous forme de collection de XTuples. Il est possible d'utiliser le caractère générique '$*$' en place du nom de collection ou du nom de la source.

Définition 4.7 : Opérateur « XSource »

Soit l'opérateur $\mathcal{S}$ défini par

$$\mathcal{S}_{s,c,R,X} : \emptyset \rightarrow \mathcal{X}$$

Il prend en entrée, un nom de source s, un nom de collection c, une requête R ainsi qu'un ensemble de chemins XPath X. L'ensemble d'arrivée est un ensemble $\mathcal{X}$ de XTuples.

Cet opérateur permet d'interroger un adaptateur *nom_source* sur une requête *requete* portant sur une collection *nom_collection*. Il renvoie en sortie le résultat de la requête effectuée par l'adaptateur sous forme de collection d'XTuple $\mathcal{C}$ dont les chemins référencés sont donnés par l'ensemble de chemins XPath *xpaths*.

L'opérateur *XSource* est un opérateur ne prenant pas de collection de XTuples en entrée.

On peut utiliser $\mathcal{S}$ des façons suivantes :
- $\mathcal{S}_R$, $\mathcal{S}_{*,R}$ ou $\mathcal{S}_{*,*,R}$ toutes les collections de toutes les sources ;
- $\mathcal{S}_{n,R}$, $\mathcal{S}_{n,*,R}$ toutes les collections de nom n de toutes les sources ;
- $\mathcal{S}_{n,s,R}$ la collection de nom n de la source s.

La figure 4.7 décrit l'opérateur XSource générant une nouvelle XRelation composée de quatre XTuples sur les XAttributs

$X = \{/personne/prenom, /personne/nom, /personne/adresse, /personne/adresse/ville\}$

Les XTuples de la XRelation sont construits au fur et à mesure du flux d'évènements SAX répondant à la requête R sur la collection C adressée à la source S. On remarquera que les nœuds peuvent être référencés plusieurs fois (XAttributs */personne/nom* des XTuples 0, 1 et 3), et qu'un XAttribut peut ne pas être présent (XAttribut */personne/adresse/rue* du XTuple 2).

L'algorithme consiste à identifier le ou les adaptateurs correspondant au nom de source spécifié et/ou gérant la collection donnée, de lui soumettre la requête et de récupérer le résultat sous forme de flux SAX. Ce flux SAX est ensuite lu à la volée pour être transformé en collection de XTuples.

```
fonction executer_source ( nom_collection, nom_source, requete_xquery)
        adaptateur := chercherAdaptateur ( nom_source, nom_collection)
        si adaptateur existe
                flux_sax := executer ( adaptateur, requete_xquery)
        fin si
       transformerEnCollectiondeXTuples ( flux_sax)
fin_fonction
```

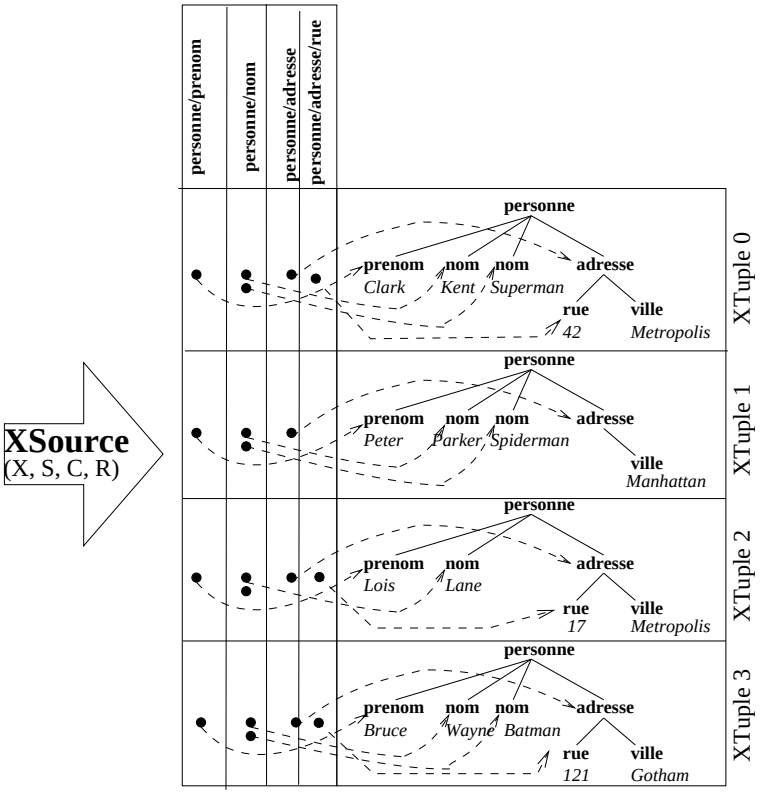


FIG. 4.7 – XSource

La transformation d'un flux SAX en XTuple se fait en analysant le flux d'évènements. Pour cela, les analyseurs SAX se basent sur des mécanismes de *rétro-appels* (*callbacks*). Au fur et à mesure que le flux d'évènements est lu, les différents évènements rencontrés déclenchent l'appel d'une fonction associée nommée *rétro-appel*. Les principaux rétro-appels à traiter sont :

- *beginDocument* : rétro-appel invoqué lors de l'évènement « début du document » ;
- *endDocument* : rétro-appel invoqué lors de l'évènement « fin du document » ;
- *beginElement* : rétro-appel invoqué lors de l'évènement « début d'un élément ». Ses paramètres sont alors entre autres, le nom de l'élément et ses attributs ;
- *endElement* : rétro-appel invoqué lors de l'évènement « fin d'un élément ». Ses paramètres sont alors entre autres, le nom de l'élément ;
- *text* : rétro-appel invoqué lors de l'évènement `#text` indiquant un texte associé à un nœud (typiquement dans le cas d'un nœud terminal).

L'algorithme permettant d'effectuer la transformation des collections de documents en XTuples se décrit suivant le pseudo-algorithme suivant :

```
declaration PileXPath chemin_courant ;

retro appel  beginDocument()  // évènement de début de document
   créer_et_initialiser_XTuple xtuple_courant
fin retro appel

retro appel  endDocument ()   // évènement de fin de document
   terminer xtuple_courant
fin retro appel


retro appel  beginElement (nom_element)  // évènement de début d'élément
   chemin_courant := empiler (chemin, nom_element)
fin retro appel

retro appel  endElement ()    // évènement de fin d'élément
    si chemin_courant est un prefixe ou egal a un XAttribut
       noeud_courant := materialiser (element, texte_courant)
       relier_arbre (noeud_courant)
    fin si
    si chemin_courant est un XAttribut
       referencer (noeud_courant)
   depiler (chemin, nom (element))
fin retro appel

retro appel  text (texte)  // évènement de texte associé à un noeud
   texte_courant :=  texte
fin retro appel
```

Le flux d'élément est lu. Un nouveau document correspond à un nouveau XTuple qui est alors créé et initialisé lors de cet évènement (`beginDocument`). Les chemins sont ensuite construit au gré du parcours préfixe de l'arbre (`beginElement`, `endElement`). Lors du parcours, les nœuds utiles (c'est-à-dire étant préfixe d'au moins un XAttribut) sont matérialisés sous forme de nœud DOM et inséré dans l'arbre en train d'être construit. Si le chemin suivi correspond exactement à un chemin d'un XAttribut, le nœud est référencé.

Propriétés de l'opérateur XSource :

- *collection d'entrées* : aucune ;
- *paramètres* : le nom de la source, le nom de la collection une requête XQuery ;
- *ordonnancement* : XSource préserve l'ordre des documents renvoyés par l'adaptateur ;
- *bloquant/non bloquant* : l'opérateur XSource est non-bloquant, et il peut commencer à construire les XTuples dès que les premiers évènements du flux SAX ont commencé à être reçus.

4.6.2.2 XProjection (π)

Dans une algèbre semi-structurée, une projection consiste à éliminer tous les nœuds (et les sous-arbres associés) ne correspondant pas à certains chemins spécifiés.

L'opérateur de XProjection généralise la projection classique aux XRelations. Il prend une XRelation en entrée et retourne une XRelation ne comportant que les XAttributs sélectionnés dans la partie attributs, et les sous-arbres non référencés sont aussi supprimés.

Définition 4.8 : Opérateur « XProjection »

Soit l'opérateur de projection π défini par

$$\pi_X : \mathcal{X} \rightarrow \mathcal{X}$$

L'opérateur ne retient que les nœuds de chaque arbre de l'ensemble de départ $\mathcal{X}$ dont le chemin appartient à l'ensemble de chemins X spécifié. L'ensemble d'arrivée est un ensemble de XTuples.

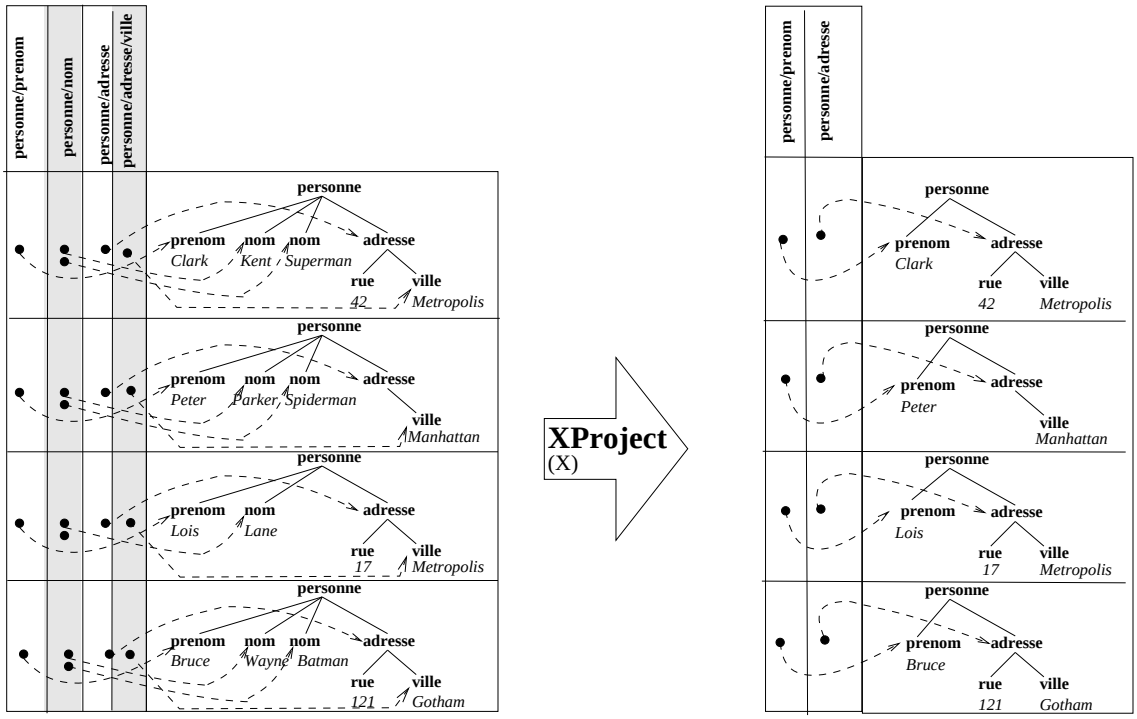


FIG. 4.8 – XProjection

La figure 4.8 décrit l'opérateur de XProjection. Il prend en argument une XRelation ainsi qu'un ensemble de chemins XPath

$$X = \{/personne/prenom, /personne/adresse\}$$

sur lequel la projection doit être faite, et rend en sortie une XRelation ne conservant que les données concernées par les XPath demandés.

Algorithme d'initialisation de l'opérateur XProjection L'algorithme d'initialisation se décompose de la manière suivante :

- identifier les colonnes dont le XAttribut ne se trouve pas parmi les chemins sur lesquels projeter ;

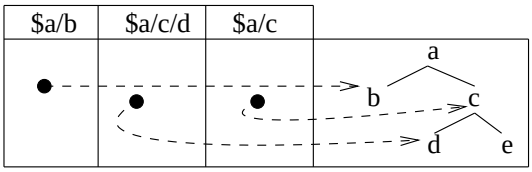


FIG. 4.9 – Exemple de XRelation

- si le XAttribut à supprimer est un sous-chemin d'un XAttribut à conserver, dans ce cas, seule la référence associée est marquée à supprimer, le sous-arbre associé doit rester intact. Exemple, sur la figure 4.9, si la projection a lieu sur $\$a/b$ et $\$a/c$, la référence $\$a/c/d$ doit être supprimée, mais le nœud associé doit malgré tout être conservé car $\$a/c$ est conservé ;
- si le XAttribut à supprimer a pour sous-chemin des XAttributs à conserver, seule la référence associée est marquée à supprimer. Exemple, sur la figure 4.9, si la projection a lieu sur $\$a/b$ et $\$a/c/d$, la référence $\$a/c$ doit être supprimée, mais le nœud associé doit malgré tout être conservé car $\$a/c/d$ est conservé ;
- sinon, si le XAttribut n'est ni sous-chemin d'un XAttribut à projeter ni a pour sous-chemin un XAttribut à projeter, alors on marque comme étant à supprimer à la fois le nœud associé et la référence.

Cet algorithme s'écrit en pseudo-algo :

```
fonction initialiser_projeter (TableauXAttributs tabxatt,
                               TableauXPath chemins_projection)
    pour chaque Xattribut xatt de tabxatt
        si xatt appartient à chemins_projection
           ne rien faire.
        sinon
            si xatt est un sous-chemin d'un XPath de chemins_projection
               marquer (xatt, DETRUIRE_REFERENCE)
            sinon si xatt a pour sous-chemin un XPath de chemins_projection
               marquer (xatt, DETRUIRE_REFERENCE)
            sinon
               marquer (xatt, DETRUIRE_NOEUD & DETRUIRE_REFERENCE)
        fin si
    fin pour

    pour chaque Noeud racine de la foret
        si aucun XPath de chemins_projection n'est sous-chemin de cette racine
            marquer (racine, DETRUIRE_RACINE)
        fin si
    fin pour
fin_fonction
```

Algorithme d'exécution de l'opérateur XProjection Une fois l'opérateur de projection initialisé, les XTuples arrivant sont traités au fur et à mesure suivant les indications d'initialisation. Les nœuds et références marqués comme tels sont détruits.

```
fonction projeter (XRelation xrel, TableauXPath chemins_projection)
   TableauXAttributs tabxatt := recupererXAttributs (xrel)
   initialiser_projeter ( tabxatt, chemins_projection)
    pour chaque XTuple xtuple_i de Collection
         pour chaque Noeud racine de la foret
         si est_marqué ( racine, DETRUIRE_RACINE)
            supprimer ( racine)
         fin si
         pour chaque Xattribut xatt_j de tabxatt
             si est_marqué (xatt, DETRUIRE_NOEUD)
                supprimer_noeud (noeud_referencé_par ( xtuple_i [j]))
             fin si
             si est_marqué (xatt, DETRUIRE_REFERENCE)
                supprimer_référence_et_colonne ( xtuple_i, j)
             fin si
         fin pour
    fin pour
fin_fonction
```

Propriétés de l'opérateur XProjection :

- *collection d'entrées* : une XRelation ;
- *paramètres* : un ensemble de XPath ;
- *ordonnancement* : l'ordonnancement est préservé puisque l'opérateur traite les XTuples au fur et à mesure ;
- *bloquant/non bloquant* : pour les mêmes raisons, l'opérateur XProjection est non-bloquant.

4.6.2.3 Restriction (σ)

Il s'agit de l'analogie de l'opérateur relationnel de restriction appliqué aux données semi-structurées.

L'opérateur XRestriction filtre chacun des XTuples d'une XRelation sur un prédicat logique donné.

Définition 4.9 : Opérateur « XRestriction »

L'opérateur de restriction σ se définit de la façon suivante.

$$\sigma_p : \mathcal{X} \rightarrow \mathcal{X}$$

Il prend en entrée un ensemble de XTuples $\mathcal{X}$, applique un prédicat p. L'ensemble d'arrivée est l'ensemble des XTuples satisfaisant le prédicat p.

La figure 4.10 décrit l'opérateur XRestriction. Il prend en entrée une XRelation ainsi qu'un prédicat logique et rend une XRelation où les XTuples ne satisfaisant pas le prédicat ont été supprimés.

Algorithme d'exécution de l'opérateur XRestriction Il n'y a pas de phase d'initialisation pour l'opérateur XRestriction.

L'algorithme de restriction suit les étapes suivantes :

1. Identifier les attributs utilisés dans le prédicat.

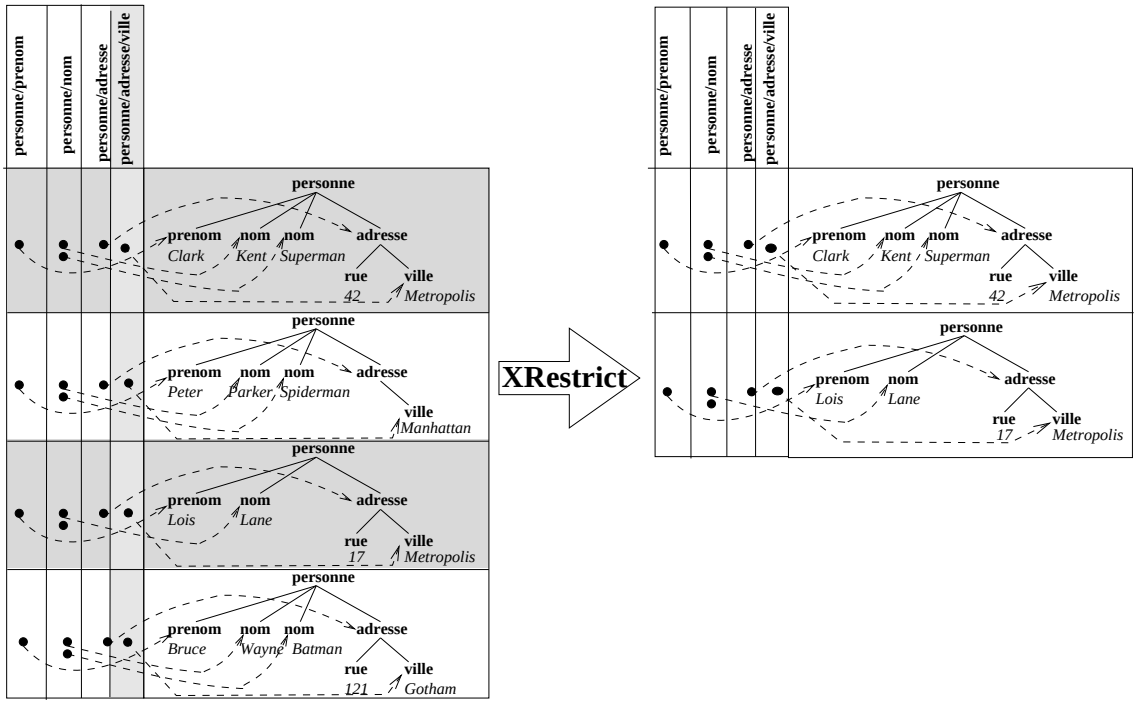


FIG. 4.10 – XRestriction

2. Tester le prédicat sur les nœuds sélectionnés. Il peut s'agir d'opérations sur les valeurs ou sur les sous-arbres.
3. Si le prédicat est vérifié alors le XTuple est conservé pour la collection résultat $\mathcal{C}_d$.

```
fonction restreindre_xtuple (XTuple xtuple)
    pour chaque Xattribut xatt de xtuple
        si xatt est dans le prédicat de restriction
           chercher le nœud référencé par cet xatt et comparer
                le sous-arbre avec l'opérateur et le sous-arbre
                donné en comparaison par le prédicat.
                 si la comparaison échoue
                    détruire le tuple entier.
                fin si
        fin si
    fin pour
fin_fonction
```

```
fonction restreindre (XRelation rel)
    pour chaque XTuple xtuple_i de rel
       restreindre_xtuple ( xtuple_i).
fin_fonction
```

Propriétés de l'opérateur XRestriction :

- *collection d'entrées* : une XRelation ;
- *paramètres* : un prédicat logique où chaque prédicat élémentaire compare un XAttribut à une constante ;
- *ordonnancement* : les XTuples sont filtrés dans l'ordre de leur arrivée, et même si certains tuples sont supprimés (s'ils ne répondent pas au prédicat), l'ordonnancement est conservé sur les XTuples résultants.
- *bloquant/non bloquant* : pour les mêmes raisons, l'opérateur est non-bloquant.

4.6.2.4 Produit cartésien ($\otimes$)

Le produit cartésien entre deux collections est une juxtaposition de chaque paire des collections.

Définition 4.10 : Opérateur « XProduit »

L'opérateur produit cartésien $\otimes$ se définit de la manière suivante :

$$\otimes : \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X}$$

Deux ensembles de XTuples sont présentés en entrée de l'opérateur. L'ensemble d'arrivée est un ensemble de XTuples.

De notre point de vue, il suffit de réaliser le produit cartésien du côté Xattributs comme le produit cartésien classique, et de réaliser une union des collections d'arbres respectifs.

Algorithme d'initialisation de l'opérateur XProduit L'initialisation de l'opérateur XProduit compare les XAttributs des XRelation à réunir.

Outre les phases d'initialisation et d'exécution, l'algorithme du produit cartésien se décompose en deux parties, l'une consiste à faire correspondre deux à deux les XTuples de chacune des XRelations, l'autre consiste à savoir comment fusionner deux XTuples.

L'initialisation de l'algorithme de fusion de deux XTuples Pour cela, un repérage des chemins communs ainsi que des contenances de chemins est réalisé dans cette phase,

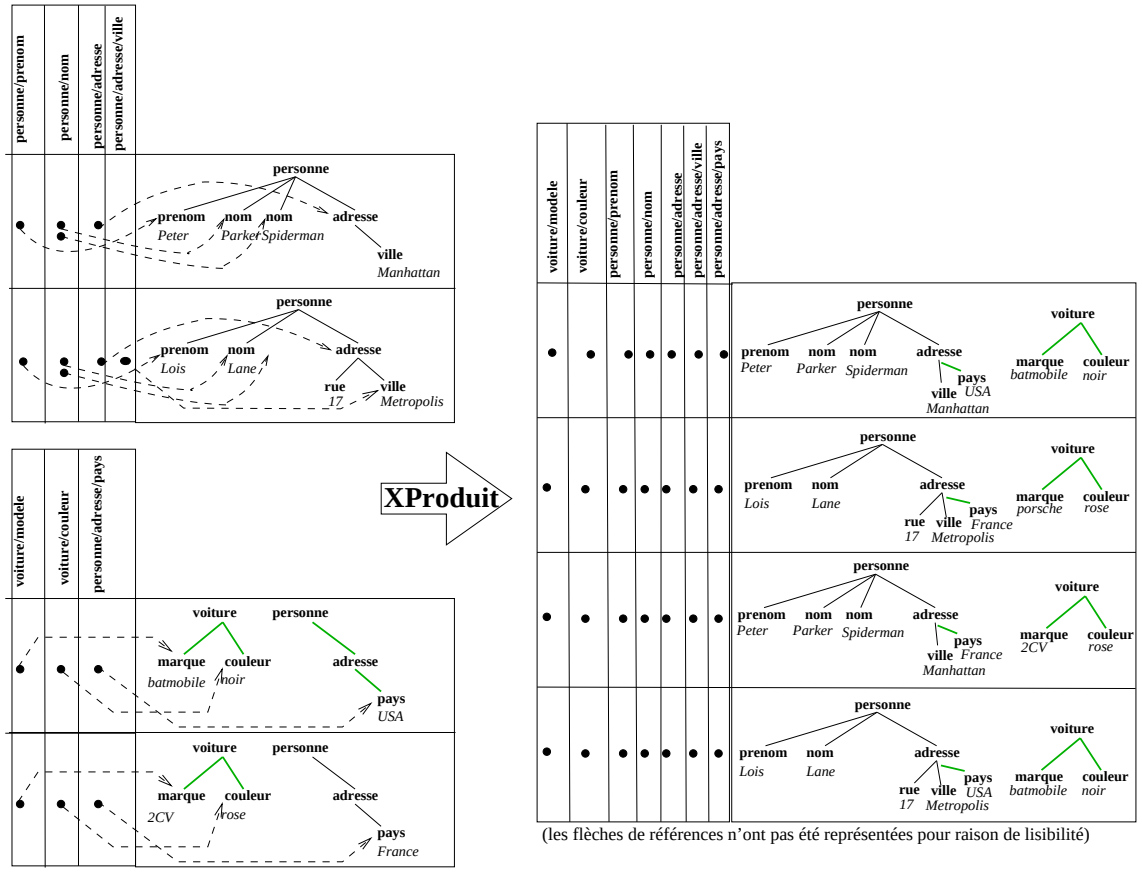


FIG. 4.11 – XProduit

et les opérations à effectuer ultérieurement sont notés.

Lors de la fusion de deux chemins $\boldsymbol{C}_1$ et $\boldsymbol{C}_2$, plusieurs cas possibles peuvent arriver (figure 4.12). Soit $\boldsymbol{C}_c := \boldsymbol{C}_1 \dot{\sqcap} \boldsymbol{C}_2$ le plus grand préfixe commun de $\boldsymbol{C}_1$ et $\boldsymbol{C}_2$.

1. $\boldsymbol{C}_c = \boldsymbol{C}_1 ET \boldsymbol{C}_c < \boldsymbol{C}_2$: Dans ce cas, le sous-chemin $(\boldsymbol{C}_2 \setminus \boldsymbol{C}_c)$ correspondant au chemin $\boldsymbol{C}_2$ sans le préfixe $\boldsymbol{C}_c$ est ajouté au chemin $\boldsymbol{C}_1$ (cf. figure 4.12 (a)).
2. $\boldsymbol{C}_c = \boldsymbol{C}_1 ET \boldsymbol{C}_c = \boldsymbol{C}_2$: Dans ce cas, les chemins $\boldsymbol{C}_1$ et $\boldsymbol{C}_2$ coïncident. Il faut vérifier si les nœuds référencés respectivement par $\boldsymbol{C}_1$ et par $\boldsymbol{C}_2$ ne sont pas en conflit de valeur. Ces nœuds peuvent être des sous-arbres, et dans ce cas, une comparaison de sous-arbre est nécessaire pour déterminer si il y a conflit. (cf. figure 4.12 (b)).
3. $\boldsymbol{C}_c < \boldsymbol{C}_1 ET \boldsymbol{C}_c < \boldsymbol{C}_2$: Dans ce cas, le sous-chemin $(\boldsymbol{C}_2 \setminus \boldsymbol{C}_c)$ correspondant au chemin $\boldsymbol{C}_2$ sans le préfixe $\boldsymbol{C}_c$ est ajouté à l'extrémité du sous-chemin $\boldsymbol{C}_c$ de $\boldsymbol{C}_1$ (cf. figure 4.12 (c)).
4. $\boldsymbol{C}_c < \boldsymbol{C}_1 ET \boldsymbol{C}_c = \boldsymbol{C}_2$: Dans ce cas, le chemin $\boldsymbol{C}_2$ est un sous-chemin de $\boldsymbol{C}_1$ et l'arbre n'est pas modifié (cf. figure 4.12 (d)).

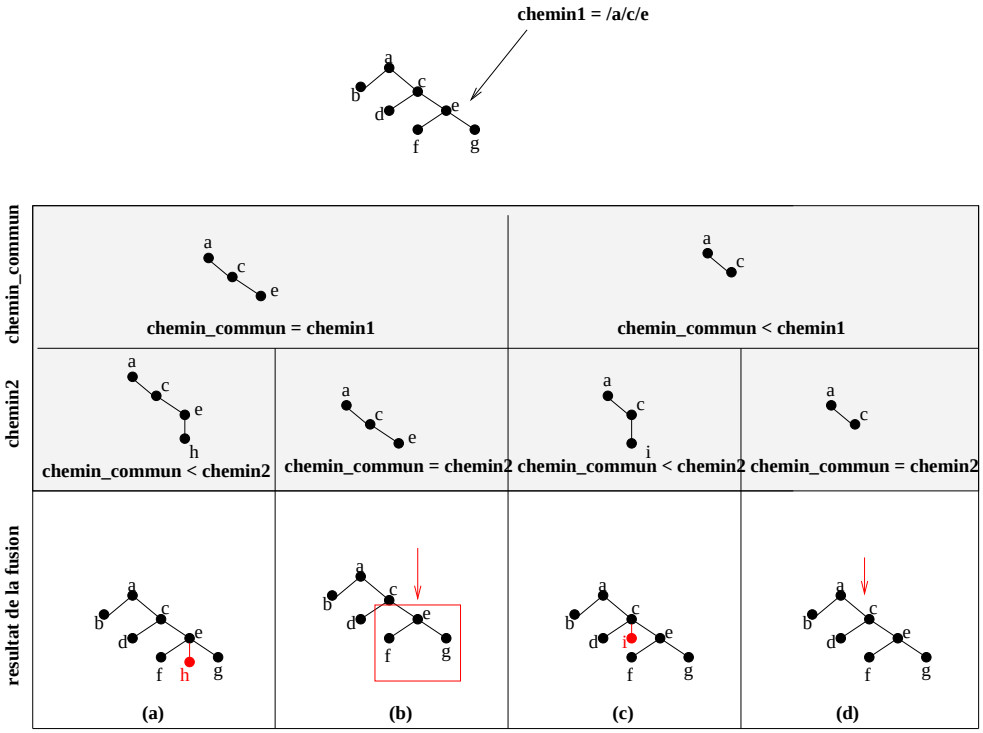


FIG. 4.12 – Les différents cas de fusion

L'algorithme s'écrit alors en pseudo-code :

```
fonction init_fusion (XMetaData xmeta1, XMetaData xmeta2
   TableauXAttributs tabxatt1 := recupererXAttributs (xrel)
   TableauXAttributs tabxatt2 := recupererXAttributs (xrel)
   Foret foret1 := recupererXAttributs (xrel)
   Foret foret2 := recupererXAttributs (xrel)


    pour chaque Noeud racine2 de foret2
        si racine2 n'existe pas dans foret1
           marquer ( racine2, AJOUTER_ARBRE)

            pour chaque Xattribut xatt_i de tabxatt2
                si xatt_i a pour préfixe racine2
                    marquer ( xatt_i, AJOUTER_REFERENCE)
                fin si
            fin pour
        fin si
    fin pour

    pour chaque Xattribut xatt2 de tabxatt2 non utilisé
       XPath chemin := plus long préfixe commun entre xatt2
                              et les sous-chemins de tabxatt1
       Xattribut xatt1 := Xattribut de tabxatt1 d'où le chemin a
                                 été extrait
        si chemin = recupererChemin ( xatt1)
            si chemin < recupererChemin ( xatt2)
               // cas (a)
               Noeud sous_arbre := extraire_arbre (foret2, chemin)
               marquer ( chemin, sous_arbre, ACCROCHER_ARBRE)
               marquer ( xatt_2, AJOUTER_REFERENCE)

            sinon // si chemin = recupererChemin ( xatt2)
               // cas (b)
               Noeud sous_arbre1 := extraire_arbre (foret1, chemin)
               Noeud sous_arbre2 := extraire_arbre (foret2, chemin)
               marquer ( xatt_2, xatt_1, COMPARER)
               marquer ( xatt_2, AJOUTER_REFERENCE)

            fin si
        sinon // si chemin < recupererChemin ( xatt1)
            si chemin < recupererChemin ( xatt2)
               // cas (c)
               Noeud sous_arbre := extraire_arbre (foret2, chemin)
               marquer ( chemin, sous_arbre, ACCROCHER_ARBRE)
               marquer ( xatt_2, AJOUTER_REFERENCE)

            sinon // si chemin = recupererChemin ( xatt2)
               // cas (d)
               marquer ( xatt_2, AJOUTER_REFERENCE)

            fin si

        fin si
    fin pour
fin_fonction
```

Algorithme d'exécution de l'opérateur XProduit Une fois la phase d'initialisation de fusion effectuée, l'exécution de l'opérateur XProduit peut s'effectuer. Pour cela, on définit la fonction $fusion$ utilisée dans l'algorithme de produit cartésien lors de son exécution. Cela consiste à appliquer les instructions calculée préalablement dans la fonction $init\_fusion$.

```
fonction fusion (XTuple xtuple_1, XTuple xtuple_2)
    XTuple xtuple_final := copier ( xtuple_1)

     pour chaque Noeud racine2 de foret2
        si est_marqué ( racine2, AJOUTER_ARBRE)
           ajouter ( xtuple_final, racine2) ;
        fin si
     fin pour


     pour chaque Xattribut xatt_j de tabxatt
         si est_marqué (xatt, ACCROCHER_ARBRE)
             si est_marqué (xatt, COMPARER)
                si comparaison (arbre1, arbre2)
                   accrocher_arbre ( chemin, sous_arbre) ;
                sinon
                   ne rien faire, sortir de la fonction fusion
            sinon
               accrocher_arbre ( chemin, sous_arbre) ;
            fin si
         fin si

         si est_marqué (xatt, AJOUTER_REFERENCE)
            ajouter_reference (noeud_referencé_par ( xtuple_i [j]))
         fin si
     fin pour
fin_fonction
```

```
Le pseudo algorithme du produit cartésien s'écrit finalement :

fonction produit_cartesien (XRelation rel1, XRelation rel2)
   init_fusion (recupererXMeta ( rel1), recupererXMeta ( rel2))

    pour chaque XTuple xtuple_i de rel1
        pour chaque XTuple xtuple_j de rel2
           fusion ( xtuple_i, xtuple_j)
        fin pour
    fin pour
fin_fonction
```

Le plus long sous-chemin commun entre deux Xattributs se fait de manière simple (comparaison de deux séquences de noms d'arcs). Le calcul est fait une seule fois et est ensuite valable pour tous les tuples.

De la même façon, chercher le plus long préfixe commun entre un chemin et un ensemble d'autres chemins, se fait simplement et une seule fois.

Propriétés de l'opérateur « XProduit » :

- *collection d'entrées* : deux XRelations ;
- *paramètres* : aucun ;
- *ordonnancement* : si l'implémentation de l'algorithme ou le paramétrage est bloquant, alors l'ordre peut être préservé. Si l'opérateur est évalué de façon non-bloquante, alors l'ordonnancement n'est pas préservé ;
- *bloquant/non bloquant* : suivant implémentation (cf. ordonnancement).

4.6.2.5 Jointure ($\bowtie$)

La jointure entre deux ensembles de XTuples correspond à une sélection entre deux attributs respectifs de ces ensembles sur le produit cartésien de ces ensembles.

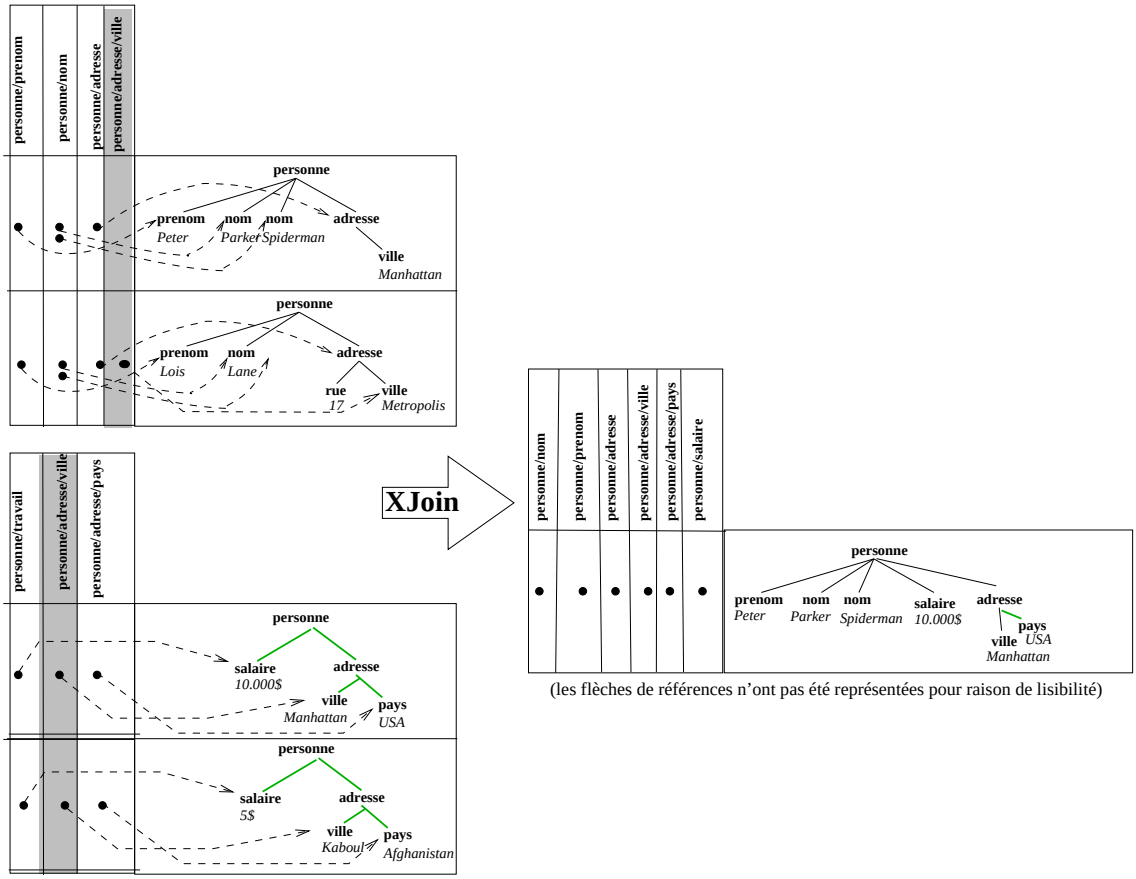
Définition 4.11 : Opérateur « XJoin»

L'opérateur de jointure $\bowtie$ se définit de la manière suivante :

$$\bowtie_p : \mathcal{X} \times \mathcal{X} \rightarrow \mathcal{X}$$

L'opérateur prend en entrée un prédicat p de sélection, et deux ensembles de XTuples. L'ensemble d'arrivée est un ensemble de XTuples.

Dans notre implémentation, nous avons utilisé l'algorithme de jointure par boucles imbriquées. Pour chaque XTuple de $\mathcal{C}_1$, on cherche dans les Xtuples de $\mathcal{C}_2$, les XTuples satisfaisants le prédicat.



(les flèches de références n'ont pas été représentées pour raison de lisibilité)

FIG. 4.13 – XJointure

```
fonction joindre (XRelation rel1, XRelation rel2, Prédicats)
   recuperer_tous_xtuples_de ( rel1)
    pour chaque XTuple xtuple_i de rel1
       n1 := chercher nœud noeud_i correspondant au chemin du predicat
                                          donné pour rel1
        pour chaque XTuple xtuple_j de rel2
          n2 := chercher nœud noeud_i correspondant au chemin du predicat
                                          donné pour rel2
          comparer (n1, n2)
           si la comparaison est vraie alors
              fusion ( xtuple_i, xtuple_j)
           fin si
        fin pour
    fin pour
fin_fonction
```

Propriétés de l'opérateur XJointure :

- *collection d'entrées* : deux XRelations ;
- *paramètres* : un critère de jointure entre les deux relations ;
- *ordonnancement* : plusieurs algorithmes peuvent être implémentés pour la XJointure dont les boucles imbriquées, le tri-fusion, et « interroger une source en dépendance avec les autres ». L'algorithme par boucle imbriquée peut être utilisé en « pipeline », les autres ne le peuvent pas. Seulemt les boucles imbriquées sans pipeline préservent l'ordre ;
- *bloquant/non bloquant* : suivant implémentation (cf. ordonnancement).

4.6.2.6 Union ($\cup$), Intersection ($\cap$), Différence ($\setminus$)

Ces opérateurs standards complètent la grammaire ainsi définie.

L'union $\cup(\mathcal{C}_1, \mathcal{C}_2) \rightarrow \mathcal{C}_d$ ajoute les tuples de $\mathcal{C}_1$ aux tuples de $\mathcal{C}_2$. C'est une opération non-bloquante. On distingue l'union avec ou sans doublons.

Propriétés de l'opérateur XUnion :

- *collection d'entrées* : deux XRelations ;
- *paramètres* : aucun ;
- *ordonnancement* : non-ordonné si non-bloquant (les tuples sont envoyés en sortie au fur et à mesure de leur arrivée) et ordonné suivant l'ordonnancement de la première collection puis suivant l'ordonnancement de la collection si bloquant ou non-parallèle ;
- *bloquant/non bloquant* : suivant implémentation (cf. ordonnancement).

L'intersection $\cap(\mathcal{C}_1, \mathcal{C}_2) \rightarrow \mathcal{C}_d$ renvoie les tuples communs à $\mathcal{C}_1$ et $\mathcal{C}_2$. C'est une opération bloquante.

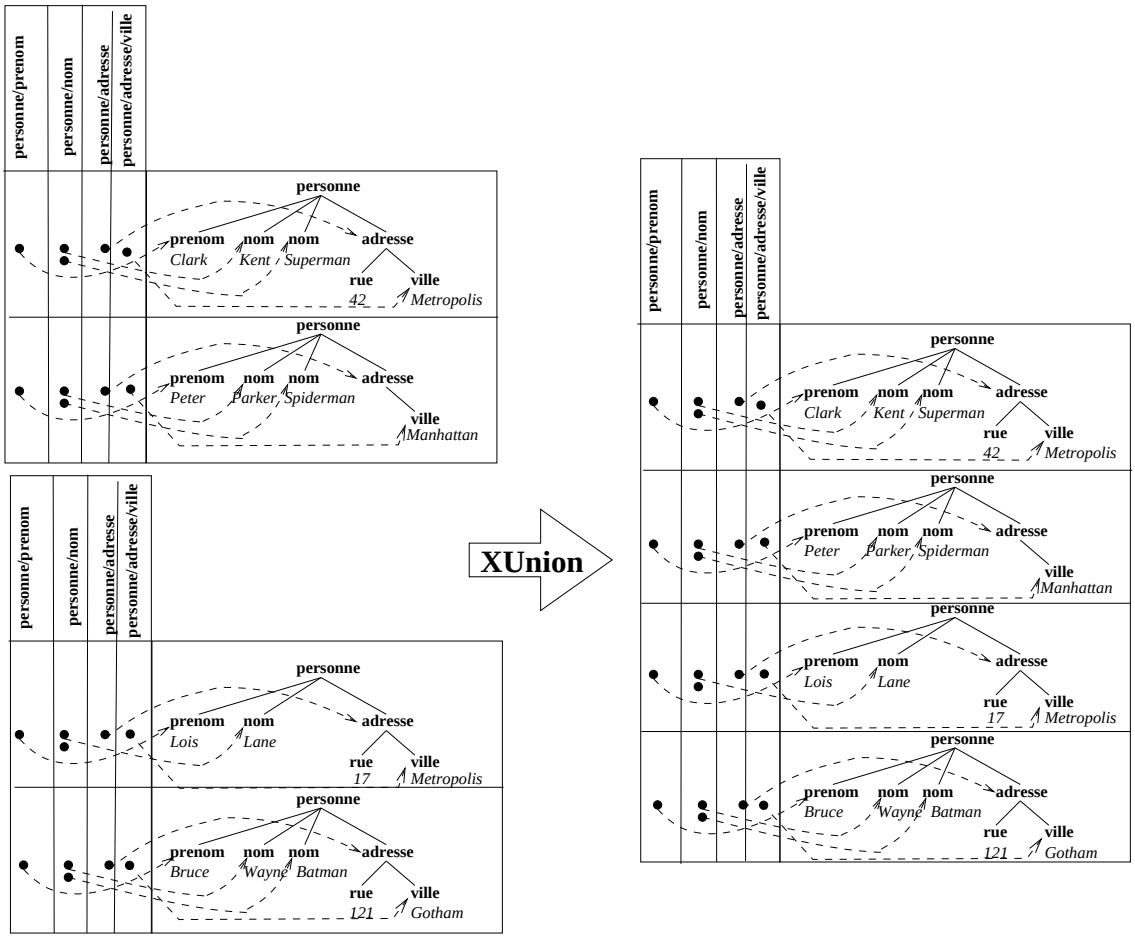


FIG. 4.14 – XUnion

Propriétés de l'opérateur XIntersection :

- *collection d'entrées* : deux XRelations ;
- *paramètres* : aucun ;
- *ordonnancement* : préservé ou non suivant implémentation bloquante ou non-bloquante ;
- *bloquant/non bloquant* : suivant implémentation.

La différence $\setminus(\mathcal{C}_1, \mathcal{C}_2) \rightarrow \mathcal{C}_d$ renvoie le complémentaire de $\mathcal{C}_2$ dans $\mathcal{C}_1$.

Propriétés de l'opérateur XDiff :

- *collection d'entrées* : deux XRelations ;
- *paramètres* : aucun ;
- *ordonnancement* : préservé ou non suivant implémentation bloquante ou non-bloquante ;
- *bloquant/non bloquant* : suivant implémentation.

4.6.2.7 Groupement, ordonnancement (γ,)

Il s'agit de déterminer suivant l'opération d'agrégation demandée la valeur répondant à cette opération en faisant intervenir l'ensemble des Xtuples de la collection. ORDER-BY est une opération utile permettant de présenter les résultats suivant un ordre déterminé. Il s'agit de trier les XTuples suivant la valeur du nœud référencé par le chemin donné en argument. Si classer des valeurs simples (chaînes, entiers, date) ne pose pas de problèmes, classer des arbres est plus problématique puisqu'il s'agit de définir un ordre sur les arbres, cette opération est normalement refusée au niveau de l'analyseur de requête et ne doit donc pas avoir lieu. L'opérateur ORDER-BY $\gamma_{attr}(\mathcal{C}_1) \rightarrow \mathcal{C}_d$ est bloquant.

Définition 4.12 : Opérateur « XOrderBy »

L'opérateur de groupement γ se définit de la manière suivante :

$$\gamma_{att} : \mathcal{X} \rightarrow \mathcal{X}$$

L'opérateur prend en entrée un nom d'attribut *att* de tri, et un ensemble de XTuples. L'ensemble d'arrivée est un ensemble de XTuples.

Propriétés de l'opérateur XOrderBy :

- *collection d'entrées* : une XRelation ;
- *paramètres* : le XAttribut sur lequel grouper ;
- *ordonnancement* : non conservé. Ordonné suivant le paramètre donné pour le OrderBy ;
- *bloquant/non bloquant* : bloquant.

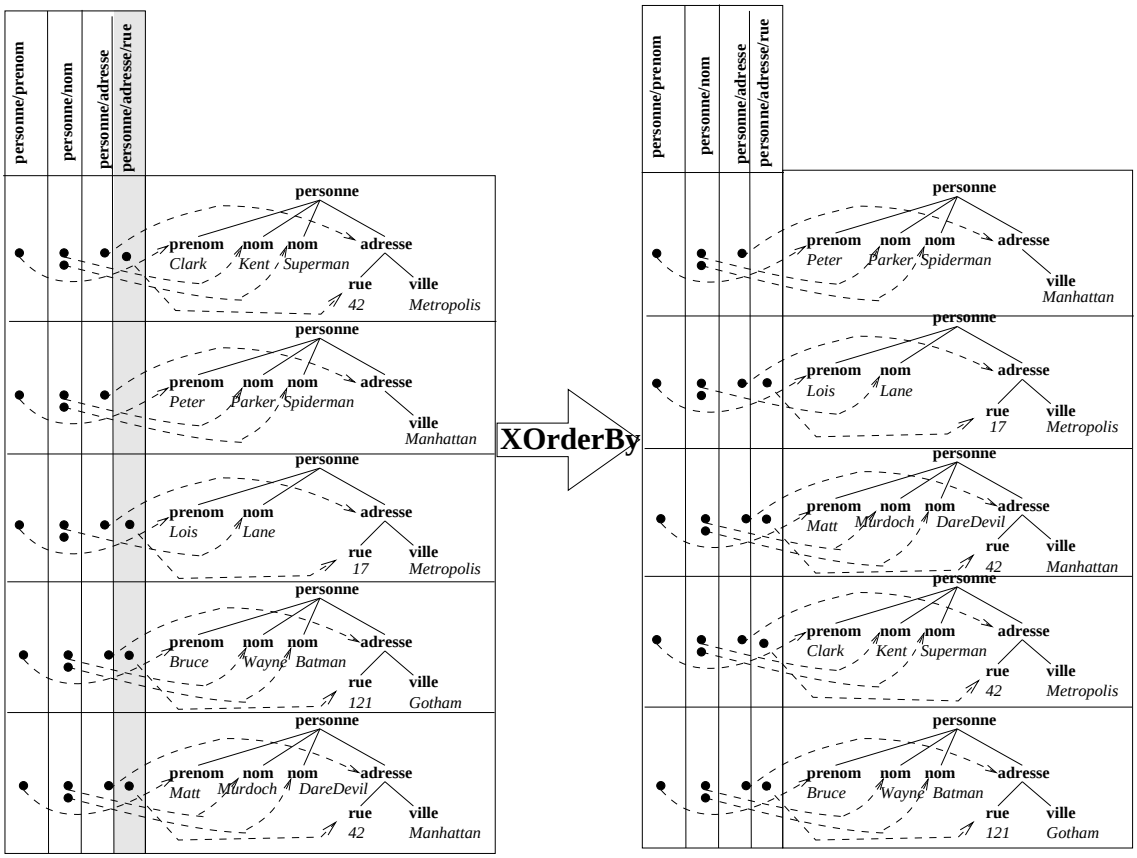


FIG. 4.15 – XOrderBy

Il s'agit d'insérer au fur et à mesure de l'arrivée des XTuples, le XTuple dans une nouvelle collection trié sur l'attribut demandé.

```
fonction order_by (XRelation rel, XPath nom_attribut)
    rel2 := nouvelle_collection
    pour chaque XTuple xtuple_i de rel
        val := recuperer_valeur_attribut ( xtuple_i, nom_attribut)
        index := chercher_place_triée ( rel2, nom_attribut, val)
       inserer xtuple_i dans rel2 à l'emplacement index
    fin pour
fin_fonction
```

Ce sont toutes des opérations bloquantes.

4.6.2.8 Agrégation ($min, max, sum, avg, count$)

Comme dans l'algèbre relationnelle, le but de l'agrégation est d'appliquer une fonction minimum, maximum, somme, moyenne ou cardinalité à une collection de valeurs. La

XRelation résultante étant composée d'un seul XTuple composé d'un arbre se réduisant à un nœud comportant la valeur calculée. À l'exception de la fonction *count* qui compte directement le nombre de XTuples, les fonctions s'appliquent sur les valeurs référencées par les XAttributs et qui doivent être typées correctement (numérique pour les fonctions d'agrégation classiques).

Définition 4.13 : Opérateur « XAgregation »

$$agreg_{att} : \mathcal{X} \rightarrow \mathcal{X}$$

où *agreg* peut être la fonction min, max, sum, avg, ou count.

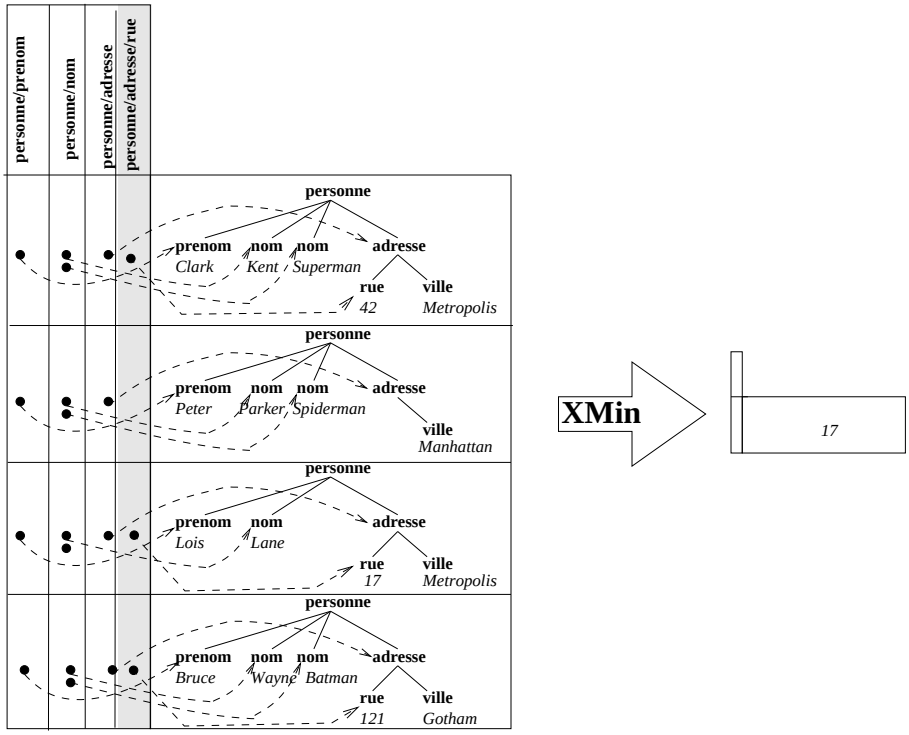


FIG. 4.16 – XAgregation (min)

```
fonction min_by (XRelation rel, nom_attribut)
    pour chaque XTuple xtuple_i de rel
    val := recuperer_valeur_attribut ( xtuple_i, nom_attribut)
        si premiere fois alors
            min := val
        sinon
            min := minimum ( min, val)
        fin si
    fin pour
   creer_xtuple (min)
fin_fonction
```

Propriétés d'un opérateur XAgrégation :

- *collection d'entrées* : un XRelation ;
- *paramètres* : l'attribut sur lequel réaliser le calcul ;
- *ordonnancement* : sans objet ;
- *bloquant/non bloquant* : bloquant.

4.7 Règles d'équivalences

Des règles d'équivalences sur les opérateurs définis précédemment peuvent être utilisées dans la génération des différents plans d'exécution. Certaines de ces règles proviennent de l'algèbre relationnelle [Codd 1972] et sont applicables, d'autres sont issues de travaux divers sur l'algèbre sur les données semi-structurées [Galanis *et al.* 2001].

4.7.1 Règles d'équivalence provenant de l'algèbre relationnelle et applicables à l'algèbre définie

Les règles d'équivalence suivantes proviennent de l'algèbre relationnelle et sont applicables à l'algèbre que nous avons utilisée.

$$\sigma_x(\sigma_y(A)) \Leftrightarrow \sigma_y(\sigma_x(A)) \tag{4.1}$$
$$\sigma_x(\sigma_y(A)) \Leftrightarrow \sigma_{x \wedge y}(A) \tag{4.2}$$
$$\bowtie_x(A, B) \Leftrightarrow \bowtie_x(B, A) \tag{4.3}$$
$$\bowtie_x(A, \bowtie_x(A, C)) \Leftrightarrow \bowtie_y(\bowtie_x(A, B), C) \tag{4.4}$$
$$\sigma_x(\bowtie_y(A, B)) \Leftrightarrow \bowtie_{x \wedge y}(A, B) \tag{4.5}$$
$$\sigma_x(\bowtie_y(A, B)) \Leftrightarrow \bowtie_{x_0}(\sigma_{x_1}(A), \sigma_{x_2}(B)) \tag{4.6}$$

La formule 4.1 traite de l'interversion possible de deux sélections successives sur un ensemble de résultats. la formule 4.2 montre qu'il est possible de regrouper deux prédicats de sélection en une seule sélection comportant pour prédicat, une conjonction de ces prédicats. La formule 4.3 parle de la commutativité de la jointure. La formule 4.4 traite de l'associativité de l'opérateur de jointure. La formule 4.5 dit qu'il est possible de regrouper une sélection suivie d'une jointure en une seule jointure avec pour prédicat la conjonction des prédicats des deux opérateurs. Et enfin la formule 4.6 énonce la distributivité de la sélection par rapport à la jointure.

4.7.2 Règles d'équivalence sur la navigation

Les règles d'équivalence suivantes sont spécifiques à la navigation dans les structures de graphes utilisées dans l'algèbre que nous avons utilisée.

$$\sigma_x(A) \Leftrightarrow \sigma_x(\phi(P)[A]), P \vdash c \tag{4.7}$$
$$\sigma_x(\phi(P)[A]) \Leftrightarrow \phi(P)[\sigma_x(A)], P \nvdash c \tag{4.8}$$

La formule 4.7 montre que si une expression de chemin apparait dans une restriction, il est possible de le suivre avant l'évaluation de la sélection. La formule 4.8 dit qu'il est possible d'intervertir les opérations de sélection et de navigation si le chemin de l'opérateur de navigation n'intervient pas dans le prédicat de l'opérateur de sélection.

Les autres règles d'équivalence définies sur l'algèbre IBM et Niagara sont de la même façon utilisables sur l'arbre algèbrique afin de l'optimiser avant de démarrer l'évaluation de l'arbre algébrique d'après le modèle de données que nous avons défini.

4.8 Conclusion

Nous avons détaillé les différents opérateurs utilisés pour la représentation de la requête, et leur mode de fonctionnement. Nous avons aussi présenté une structure permettant de manipuler les données semi-structurées de façon aussi simple que les données relationnelles dans une évaluation de requêtes.

Certaines algèbres se basent entièrement sur les graphes ou les structures arborescentes. D'autres convertissent les requêtes semi-structurées en relationnel et ne travaillent que sur des données relationnelles ; elles ne travaillent pas efficacement sur les chemins et les structures arborescentes.

Notre approche est mixte : elle utilise des opérateurs relationnels étendus pour la performance tout en manipulant des données de type arborescent enrichies de pointeurs indexés sous forme tabulaire.

La connexion des opérateurs de notre algèbre à des entrées/sorties flux permet de s'adapter aisément aux flux SAX. En utilisant les propriétés non-bloquantes de certains opérateurs, des évaluations parallèles de flux ont pu être mises en œuvre. L'ordonnancement des nœuds est conservé (traités au fur et à mesure dans le flux).

L'originalité de notre modèle se base sur la manière d'adapter les opérateurs relationnels à une structure de données semi-structurées en conservant un comportement similaire. Ainsi, la plupart des travaux en terme d'optimisation, et d'implémentation faits dans les architectures de médiation à base de données relationnelles restent valables. La conservation des graphes et des pointeurs permettent d'optimiser le processus de reconstruction. L'algèbre de type NIAGARA s'adapte ensuite aisément au modèle de données que nous avons défini, de sorte à pouvoir réutiliser les règles d'équivalence qui ont été établies.

L'algèbre la plus proche de la XAlgèbre est l'algèbre TAX [Jagadish *et al.* 2001]. Cette algèbre propose en effet de la même façon de manipuler des structures d'arbres à l'aide d'opérateurs relationnels étendus. Les forêts d'arbres résultats sont de la même façon considérés comme des tuples. La majeure différence entre leur approche et la notre est qu'ils utilisent des arbres de motif (*pattern tree*) pour indiquer la position des nœuds à traiter dans la forêt d'arbres XML. Si cette méthode permet de ne pas avoir à extraire tous les ensembles de chemins nécessaires à l'évaluation lors de la création de l'arbre algébrique, elle a pour inconvénient de devoir calculer et faire construire pour chaque opérateur, lors de l'évaluation, un arbre témoin (*witness tree*) référençant les nœuds de l'arbre résultat. Cette approche peut s'avérer coûteuse. Dans notre approche, le référencement n'est effectué qu'une seule fois pour toutes (par les XSources).

La construction du résultat final suivant la structure donnée dans la clause « CONSTRUCT », ne fait pas l'objet d'opérateurs algébriques. En effet, l'opération de construction du résultat se fait par un module indépendant de recomposition. Le recomposeur remplace les variables liées (*binding variables*) par les valeurs trouvées au fur et à mesure de l'évaluation. Un mécanisme de balisage créé la structure finale au fur et à mesure et insère les valeurs des variables résolues Ce module pourrait être intégré sous forme d'opérateurs algébriques XRegroup et XSplit afin de pouvoir être intégré dans l'arbre d'exécution et faciliter les reconstructions imbriquées.

Chapitre 5

Modèle de coût pour médiation de données semi-structurées

5.1 Introduction

Dans le chapitre précédent, nous avons montré que l'architecture du médiateur comportait un optimiseur de requêtes. En effet, pour une requête donnée, il peut y avoir plusieurs façons d'exécuter cette requête. Il s'agit donc de modéliser le coût d'exécution de chacun de ces plans d'exécution afin de choisir le plus optimal. Déterminer le coût d'un plan d'exécution peut s'avérer complexe, surtout dans le cadre d'une architecture faisant intervenir des sources hétérogènes semi-structurées.

Cela nous amène à nous intéresser aux problèmes suivants :

1. Quel est le modèle permettant au mieux d'estimer simplement le coût d'une source de données semi-structurées ?
2. Comment communiquer ces informations de coûts au médiateur, en tenant compte de l'aspect semi-structuré que peuvent prendre certaines sources ?
3. Comment le médiateur peut-il intégrer les informations de coûts des différents adaptateurs ?
4. Comment le médiateur calcule-t-il le modèle de coût d'opérations exécutées à son niveau ?

Pour répondre au point ①, nous allons illustrer le fonctionnement d'une source semi-structurées en décrivant le prototype ReposiX [Yeh et Dang-Ngoc 2001] que nous avons prototypé dans le cadre du projet MUSE. ReposiX est un prototype de stockage natif de données semi-structurées interrogeable *via* des primitives de type XPath. Nous estimerons le modèle de coût d'un tel SGBD natif.

Dans sa thèse, [Naacke 1999] montre comment le prototype DISCO répond aux points ②, ③ et ④ pour des données relationnelles ou objets. Nous allons étendre ces solutions aux données semi-structurées. Nous allons pour cela s'interroger sur les statistiques utiles et les formules de coût à exporter dans le cadre des données semi-structurées, et la façon dont on doit les exporter. Nous montrerons comment intégrer ces données dans le calcul de coût du médiateur. Ensuite, nous évaluerons le coût des opérateurs de l'algèbre que nous avons défini dans le chapitre précédent afin de déterminer le coût de l'exécution d'un plan sur le médiateur.

5.2 Plan du chapitre

Nous présentons tout d'abord un état de l'art sur les modèle de coût dans les architectures de médiation (section 5.3). Nous discuterons ensuite des différents éléments utilisés dans le calcul de modèle de coût dans la section 5.4. Nous étudierons ensuite dans la section 5.5 un modèle de coût d'un adaptateur de données semi-structurées en prenant pour source d'étude le système natif ReposiX développé au sein du projet MUSE. Enfin nous définirons dans la section 5.6 le modèle de coût initial du médiateur ainsi que le modèle de coût associé aux différents opérateurs défini par le médiateur. Nous verrons comment s'intègre toutes ces différentes formules de coût pour l'estimation du coût d'un plan d'exécution. Enfin nous conclurons dans la section 5.7

5.3 État de l'art

Il peut y avoir plusieurs, voire une infinité de manières de traiter une requête ; chacune de ces façons constitue un *plan d'exécution*. L'ensemble des *plans d'exécution équivalents* (c'est-à-dire donnant le même résultat pour une requête donnée) forme *l'espace des possibilités*. La cardinalité de cet ensemble pouvant être très grande voire infinie, il est impossible d'explorer *toutes* les possibilités afin de déterminer le *plan optimal*. Il s'agit donc de trouver des heuristiques afin de restreindre cet ensemble en un ensemble fini beaucoup plus petit appelé *espace de recherche*.

Le rôle de l'*optimiseur* est de déterminer l'espace de recherche, puis d'examiner chacune des possibilités d'exécution de cet espace afin de choisir le plan optimal.

Le *coût* d'un modèle d'exécution peut s'exprimer en terme de *temps d'exécution* (temps observé entre le lancement de la requête jusqu'à l'obtention des résultats), de *travail* (consommation de ressources : communications, place mémoire), ou encore d'*unités monétaires* (prix des requêtes et des communications).

La manipulation de sources hétérogènes implique des traitements différents entre

chacune des sources. De ce fait, les algorithmes classiques d'optimisation utilisés dans les bases de données ne peuvent pas tous s'appliquer dans le cas d'une optimisation sur des sources de données hétérogènes - ceci à cause des mauvaises voire de l'absence de connaissances sur les données manipulées (index, propriétés des ressources, schémas, cardinalités).

Pour une requête donnée, les problèmes qui se posent lors de l'évaluation sont :

- *comment déterminer l'espace de recherche* : Comment déterminer l'ensemble des plans d'exécution possibles et qu'il est raisonnable d'étudier ?
- *comment déterminer le coût d'un plan d'exécution* : Comment quantifier chaque plan d'exécution ?
- *comment choisir le plan d'exécution optimal* : En fonction des deux points cités ci-dessus, comment trouver le plan de coût optimal ?
- *comment regrouper l'information* : Savoir comment classer, restructurer et composer les résultats récupérés.

5.3.1 Estimation des coûts

L'estimation des coûts se fait principalement suivant deux critères : soit en tenant compte de la consommation des ressources. Dans le premier cas, cela revient à évaluer le traitement sans tenir compte du parallélisme éventuel, soit en ne tenant compte que du traitement des ressources. Dans ce second cas le parallélisme est pris en considération.

Parmi les critères pris en considération lors du calcul du coût d'une requête, nous pouvons considérer : le coût de *communication*, le coût de *traitement du médiateur*, le coût des différents *traitements des adaptateurs*, et enfin le *coût de congestion* du réseau.

coût_total = coût_communication + coût_médiateur + coût_adaptateurs

soit dans l'ordre du plus au moins coûteux :

- *coût_communication* : c'est le temps de communication entre le médiateur et les adaptateurs. Ce coût est lié à la taille des données transférées (taille de la requête, cardinalité du résultat et taille d'un résultat), mais aussi au paramétrage même du réseau (débit de communication, protocole utilisé). Le coût de communication entre le médiateur et un adaptateur peut être estimé comme suit :

$$\begin{aligned} cout\_communication &= temps\_d'initialisation + temps\_de\_transmission \\ &= temps\_d'initialisation + \frac{nombre\_d'octets\_a\_transmettre}{debit\_en\_octets/secondes} \end{aligned}$$

où *temps_d'initialisation* est le temps d'établissement de la connexion auquel on

ajoute *temps_de_transmission* le temps de transmission des données.

- *coût_médiateur* : c'est le temps que le médiateur met à traiter la requête et à traiter les différentes réponses renvoyées par les adaptateurs avant de donner le résultat final. Ce coût dépend des caractéristiques de la machine sur laquelle tourne le médiateur (CPU, mémoire), mais surtout de la façon dont sont implémentés les différents calculs sur les données (opérateurs de jointure-sélection-projection, reconstruction du résultat) au sein du médiateur.

$$\begin{aligned} cout\_mediateur &= temps\_CPU \\ &= temps\_par\_instruction \times nombre\_d'instructions \end{aligned}$$

- *coût_adaptateurs* : c'est le temps que les adaptateurs mettent à répondre à une sous-requête qui leur a été envoyée par le médiateur. Ce coût est lié au traitement de la requête par l'adaptateur (conversion de la requête et des résultat), mais surtout au coût de traitement de la source sous-jacente (ex : temps d'exécution d'une requête sur un SGBDR, recherche de motifs dans un document web). Ce dernier dépend de la requête à traiter. Le coût d'un adaptateur pour une requête donnée peut être estimé comme suit :

$$\begin{aligned} cout\_adaptateur &= temps\_CPU + temps\_entrees\_sorties \\ &= temps\_par\_instruction \times nombre\_d'instructions \\ &\quad + temps\_d'une\_entree/sortie \times nombre\_d'entrees\_sorties \end{aligned} \tag{5.1}$$

Dans le cas d'un réseau comme l'Internet (WAN), les temps de communication a une importance considérable, de sorte que dans le calcul du coût, on peut pratiquement négliger le temps CPU et le temps d'entrées/sorties.

Type de disque	Temps d'entrees/sorties
USB 1	1.5 Mb/s
USB 2	10 Mb/s
FireWire	30 Mb/s
EIDE (Ultra DMA66)	66 Mb/s
Ultra SCSI 160	160 Mb/s

Dans le cas d'un réseau local (LAN), les temps de communication se situent entre 100 mb/s (12 Mb/s) et 1 gb/s (128 Mb/s). Les temps de communication et les temps de traitement local sont plus ou moins équivalents.

Dans un modèle centralisé, l'évaluateur de requête de la base a une connaissance précise des sources qu'il administre, le plus souvent par l'intermédiaire de fonctions *internes* qui ont été implémentées lors de sa conception, et dans ce cas la formule (5.1) peut être utilisée et calculé par approximation.

Cela n'est malheureusement plus le cas dans le cadre d'un système hétérogène, où les sources sont autonomes et ne communiquent pas *a priori* leurs paramètres de coût.

Pour remédier à cela, différentes approches dans les calculs de modèles de coût ont été étudiées :

- *la méthode par calibration* : [Du *et al.* 1992] Il s'agit de rechercher des requêtes-types afin de déterminer les constantes de la source à étudier, et d'estimer les paramètres la régissant. Cette méthode a par la suite été affinée (échantillonnage [Q.Zhu et Larson 1998], s'adaptant à l'environnement [Zhu *et al.* 2000b] [Zhu 1995]) et étendue a d'autres modèles de données ([Gardarin *et al.* 1996b]) ;
- *la méthode par historique* : [Adali *et al.* 1996] Cette méthode s'appuie sur les statistiques des requêtes précédentes tant en unités de temps qu'en cardinalité des résultats obtenus ;
- *la méthode de coût défini par les adaptateurs* : [Haas *et al.* 1997] Le coût des sous-requêtes traitées par les adaptateurs et leur source doit être estimé par un modèle de coût défini séparément par chaque adaptateur ;
- *le méthode de coût adaptatif* : [Zhu 1995] Le modèle de coût adaptatif s'appuie sur l'évocation périodique d'échantillons. Cela permet de prendre en considération les variations de l'environnement. DISCO [Naacke *et al.* 1998] étend ce modèle en proposant une hiérarchie de modèle de coût pouvant utiliser aussi bien un modèle de coût par défaut que les coûts par calibration, par échantillonnage et par historique.

5.3.1.1 le coût par calibration

Le modèle de coût par calibration [Du *et al.* 1992] a été tout d'abord défini dans un système hétérogène ne comportant que des sources relationnelles lors du projet PEGASUS [Ahmed *et al.* 1987]. Il a ensuite été étendu au modèle objet [Gardarin *et al.* 1996b] dans le projet IRO-DB [Gardarin 1997].

L'estimation initiale repose sur les hypothèses suivantes :
- la taille du tuple est considérée comme fixe ;
- il y a exactement une condition de jointure/sélection dans une relation ;
- le tuple entier est projeté en résultat ;
- tous les attributs sont des entiers.

Avec deux relations R_1 et R_2 avec n_1 et n_2 tuples, on peut estimer le coût des opérations. Soit :

- c_0 le coût initial : traitement de la requête et initialisation de la sélection. Ce composant dépend du SGBD mais est indépendant de la relation ou de la requête ;
- c_1 le coût pour trouver un tuple satisfaisant le critère de sélection : coût des entrées/sorties (c_1^{io}) et les surcharges induites par le verrouillage et l'appel itératif

(c_1^{cpu}) ;
- c_2 le coût pour traiter un tuple sélectionné.

Muni de ces définitions, il est proposé de formaliser le coût d'une sélection séquentielle sur r_1 avec la sélectivité S_1 :

$$CS_{ss}(R_1) = [c_0(R_1)] + [\underbrace{(c_1^{io}(R_1) + c_1^{cpu}(R_1))}_{c_1(R_1)} \times n_1] + [c_2(R_1) \times n_1 \times S_1] \tag{5.2}$$

Une formule similaire est aussi proposée par les auteurs afin d'évaluer une sélection avec index ou avec cluster.

Le coût d'une jointure sur R_1 et R_2 avec une sélectivité J_{12} est estimé par :

$$\begin{aligned} CJ_{nl}(R_1, R_2) = {} & CS_{ss}(R_1) + c_0(R_2) + c_1^{io}(R_2) \\ & + (n_1 \times S_1 \times (CS_{ss}(R_2) - c_0(R_2) - c_1^{cpu}(R_2))) \end{aligned} \tag{5.3}$$

Procédure de calibration Pour calibrer une table R_n, on fait varier les colonnes C_i dans les requêtes :

$$select \quad R_n.C_i \quad from \quad R_n \quad where \quad C_i = c$$

et

$$select \quad R_n.C_i \quad from \quad R_n \quad where \quad C_i < c$$

La base de calibration est installée au sein du système source.

Pour chacune de ces requêtes, le coût $CS_{ss}(R_i)$ est calculé. Puis, le résultat est analysé pour obtenir la sélectivité de la requête S_1. Enfin, la cardinalité de la table accédée n_1 est récupéré. La variation de requêtes sur trois tables R_n différentes dans l'équation (5.2), permet de générer trois équations à trois inconnues c_0, c_1 et c_2 qui sont résolues afin de déterminer les valeurs de ces constantes.

Une validation pour la formule de jointure peut être réalisée de la même façon à l'aide du modèle de requête suivant :

$$select \quad R_n.C_i, R_m.C_j \quad where\ R_n.C_i = c \quad and \quad R_n.C_i = R_m.C_j$$

en paramétrant la formule (5.3).

La méthode présente les inconvénients suivants :

1. Il est difficile de prédire la façon dont la requête va être exécutée par la source. En effet, les index, le choix d'ordonnancement des jointures, placement des données et pagination ne sont pas forcément connus. De sorte que la formule de coût à utiliser ne peut être déterminée (sélection séquentielle, sélection avec index, sélection avec clustered index). Ceci peut entraîner des erreurs lors de la détermination des constantes et lors de l'évaluation du coût d'un plan.
2. La méthode de calibration ne tient pas compte des variations de l'environnement d'exécution de la requête.
3. Il n'est pas toujours possible de créer une base de calibration sur la source elle-même (droit d'accès, autonomie de la source).
4. Il faut pouvoir déterminer les requêtes-types afin de couvrir le maximum de cas. Ceci n'est pas toujours évident.

Approche par échantillonnage. Afin de pouvoir résoudre le problème ① (prédiction de la manière dont va être évaluée la requête par la source), [Q.Zhu et Larson 1998] propose une méthode d'échantillonnage. La procédure consiste à :

- organiser les requêtes en classes, une classe correspondant à un type de requête ;
- générer une requête-type pour chaque classe ;
- envoyer la requête sur la source ;
- générer une formule de coût pour chaque classe ;
- affiner la formule de coût de chaque classe par l'application de multiples régressions [Zhu et Larson 1996].

Le problème difficile est de déterminer les classes de requêtes ; il s'agit de prendre en compte les éléments suivants :

- *le type de requête* : requête unaire, jointure, etc.
- *la cardinalité de la table* : le nombre de tuples, les colonne indexées, etc.
- *les caractéristiques de la source* : les méthodes supportées, etc.

En résumé, cette méthode consiste à classifier un ensemble de requêtes, puis par extraction d'échantillons et de jeux d'essai, déterminer par des calculs de régression les formules caractérisant le coût des requêtes d'une source.

Approche qualitative. Dans son article, [Zhu *et al.* 2000b] propose une solution au problème ② (variation de l'environnement) : le coût de la requête peut changer très significativement dans un environnement dynamique. On distingue trois grands types de changement :

1. Changement peu fréquents : fréquence du processeur, type et version du SGBD ; ceci ne cause en général pas de problèmes.

2. Changement occasionnel : configuration de la base, mémoire physique, placement des données ; une solution serait de reconstruire périodiquement le modèle de coût.
3. Changement fréquent : charge CPU, E/S par secondes ; il est impossible de reconstruire le modèle de coût lors de changements trop fréquents. On doit de plus tenir compte de la difficulté d'inclure des variables dynamiques (trop nombreuses, à interactions inconnues).

L'approche qualitative consiste à :
- examiner l'effet combiné de tous les facteurs sur le coût ;
- utiliser le coût d'une requête exploratoire pour mesurer le niveau de contention ;
- diviser le niveau de contention en états finis (pas de contention, un peu, moyennement, beaucoup) ;
- utiliser une variable qualitative dans le modèle de coût pour indiquer le niveau de contention.

Analyse fractionnelle et probabiliste. Une grosse (coûteuse) requête peut traverser différents état pendant son exécution. Pour cela [Zhu *et al.* 2000a] propose de définir les différents états possibles, et de calculer les probabilités de transitions d'un état à un autre. Par l'utilisation de chaînes de Markov on définit ainsi le coût à :

$$C(Q) = \frac{1}{\sum_{i=1}^{M} \frac{\Pi_i}{C(Q,S_i)}}$$

où $\Pi_j = lim_{n\to\infty} P_{ij}(n)$ la probabilité limite où P_{ij} est la probabilité de passer d'un état S_i à un état S_j après n étapes.

De cette façon, il est possible de gérer un environnement évoluant rapidement et de façon aléatoire.

Extension au modèle objet. Ce principe de calibration a été étendu au modèle objet [Gardarin *et al.* 1996b]. Les formules de coût sont similaires à celles données précédemment à l'exception de la différenciation entre le nombre d'objets et le nombre de pages dans une collection, ainsi que de l'introduction d'un coût de projection. Afin de pouvoir intégrer la notion de navigation dans le processus de requête, l'introduction du coût de *suivi de pointeur* (*pointer chasing*) a également été introduit [Gardarin *et al.* 1996a].

5.3.1.2 le coût par historique.

Une autre approche est une approche statistique [Adali *et al.* 1996] basée sur l'historique des requêtes déjà exécutées. Pour une requête executée, un vecteur de coût est rempli avec les coefficients suivant : $vecteur\_cout = [T_F, T_A, Card]$, avec :

- T_F : temps pour obtenir le premier résultat ;
- T_A : temps pour obtenir tous les résultats ;
- $Card$: cardinalité de la réponse.

Ce vecteur de coût est ensuite enregistré dans une base d'historique locale sous forme du triplet $(domaine, vecteur\_cout, date\_courante)$ où :

- $domaine$: $domaine : fonction(arg_1, .., arg_n)$;
- $vecteur\_cout$: le vecteur de coût créé ;
- $date\_courante$: date à laquelle l'appel a été enregistré dans la base.

La notion de domaine de requête est présentée en primeur dans cet article. Elle se base sur l'inclusion du domaine de résultats d'une requête R1 dans le domaine de résultats d'une autre requête R2. Pour cela un *cache* de requêtes a été défini. Pour estimer le coût d'une nouvelle requête, celle-ci est comparée avec les prédicats contenus dans la base locale d'historique.

Cette méthode est appropriées dans le cas où le médiateur a accès à une base de données locale pour stocker les historiques de coût et si le type de requêtes ne diffèrent pas beaucoup d'une requête à l'autre.

5.3.1.3 le coût défini par les adaptateurs

L'article de [Haas *et al.* 1997] fait remarquer qu'il n'y a pas de modèle de coût générique pour tous les adaptateurs (comme défini précédemment), chaque adaptateur ayant son modèle de coût.

Cette méthode expérimentée dans le projet GARLIC s'appuie sur une connaissance *par défaut* des adaptateurs contrôlant une source. Elle utilise aussi le plus petit modèle commun de coût. Le médiateur GARLIC propose un modèle de coût générique par défaut des adaptateurs qu'il manipule et qui peuvent être personnalisés par le programmeur de l'adaptateur. Un plan est un arbre de *Plan OPerator* (POP), où chaque POP est un nœud possédant un ensemble de propriétés décrivant le rôle de l'opérateur associé (collection accédée, prédicat, attributs projetés), son coût, et le nombre de tuples renvoyés. Pour chaque opérateur (POP), on ne considère pour le calcul du coût que les trois paramètres suivants :

- le coût total C_T : le temps en seconde pour exécuter l'opérateur et obtenir l'intégralité du résultat ;
- le cout de ré-exécution C_R : le coût pour re-exécuter le POP une seconde fois ;
- la cardinalité estimée du POP $Card$.

La formule de coût est la suivante :

$$Cout = C_T + C_R \times (Card - 1)$$

Ces trois informations suffisent pour estimer le coût d'une requête. De plus, elles peuvent être obtenues facilement pour une grande variété de traitement effectués par une source. Ces trois informations fournissent un niveau d'abstraction bien adapté pour l'intégration des sources très hétérogènes (SGBD-R, serveur d'images).

Modèle de coût générique. L'estimation du coût est bien adaptée pour les bases de données classiques, mais cela est plus difficile dans les bases de données hétérogènes [Naacke *et al.* 1998]. Certaines sources ne peuvent exporter leurs informations de coût. Dans DISCO, [Naacke *et al.* 1998] propose donc d'utiliser un modèle de coût générique pour les sources n'exportant pas d'information de coût tout en utilisant les modèles de coût des sources ayant la possibilité de l'exporter.

Cette solution très proche de celle de GARLIC, comporte en plus un langage d'exportation d'estimation de coût (formule de coût et statistiques). La syntaxe des formules de coût se présente sous la forme d'une expression mathématique classique. Une formule peut référencer une statistique par l'intermédiaire d'une variable.

Soit par exemple la règle suivante définie dans le langage d'exportation de coût de DISCO :

```
1     scan (article) :-
2         rem: acces aux statistiques
3         totalSize (article, Taille),
4         Count (article, Card),
5         SS0 (Init),
6         SS1 (Process),
7         SS2 (Transmit),
8
9         rem: formules de coût
10        TimeFirst = Init
11        TotalTime = Init + Process * Taille + Transmit * Card
```

Cette règle définit le coût d'une lecture séquentielle (`scan`) de la collection `article`. Dans les lignes 3 à 7, les variables `Taille`, `Card`, `Init`, `Process` et `Transmit` se voient affectées des statistiques correspondantes. Les formules de coût utilisant ces variables sont ensuite définies dans les lignes 10 et 11.

Les techniques d'évaluation de coût de DISCO sont aussi enrichies par une classification hiérarchique des modèles de coût. Cette hiérarchie offre plusieurs niveaux de spécialisation des formules. Elle permet de surcharger les formules de coût génériques par d'autres formules plus spécifiques. Cette hiérarchie permet d'intégrer des modèles de coût

hétérogènes pour une grande variété de sources.

Par le modèle de coût générique, DISCO regroupe les différentes méthodes de modélisation de coût en généralisant l'approche par calibration et celle par échantillonnage. En effet, cette approche permet d'une part de décrire les statistiques et les formules de coût au niveau des adaptateurs, et d'autre part, une façon cohérente d'enrichir le modèle de coût du médiateur avec ces informations.

5.3.1.4 Adaptation du modèle de coût requêtes comportant des chemins

Par rapport aux données relationnelles organisées de façon tabulaire et comprenant éventuellement des index sur des attributs, les données se présentant sous forme arborescente –c'est-à-dire les données objets ou semi-structurées– ont un coût plus difficilement modélisable. En effet, le concept de *chemin* est une nouvelle notion introduite pour les données objets et reprises pour les données semi-structurées.

Il existe trois façons d'aborder la traversée de chemin :

- « en profondeur d'abord » (*Depth-First-Fetch* (DFF)) : on suit le chemin depuis la racine jusqu'à la collection cible en utilisant une traversée de graphe en profondeur d'abord. L'avantage est qu'on ne génère pas de résultats intermédiaires et cette méthode est très efficace si la mémoire est assez grande pour contenir tous les arbres ;
- « en largeur d'abord » (*Breadth-First-Fetch* (BFF)) : on traite l'arbre en utilisant l'algorithme de *jointure directe Forward Join* (FJ) basé sur le suivi de pointeurs entre deux collections. Cette stratégie est aussi nommée *top-down*. Par exemple, en reprenant le document 4.1, si l'on cherche toutes les personnes de la liste dont l'âge est inférieur à 30, en utilisant cette stratégie, on chercherait toutes les *personne* de *liste*, puis, on chercherait tous les objets *age* reliés à *personne*, et enfin, on sélectionnerait seulement les éléments *age* satisfaisant le prédicat < 30 ;
- « en largeur d'abord inversé » (*Reverse-Breadth-First-Fetch* (RBFF)) : c'est une séquence de jointures binaires entre deux collections proches pour suivre le chemin, mais on utilise l'ordre inverse du chemin. Chaque jointure est donc appelée *jointure inversée* (*Reverse Join* (RJ)). Cette stratégie est appelée aussi *bottom-up* et la requête précédente se résoudrait -en adoptant cette stratégie- en identifiant tous les objets dont la valeur satisfait le prédicat < 30 et dont l'étiquette est *age*, puis on chercherait tous les parents de chacun des résultats étiquetés *personne*, puis tous les parents de ces résultats étiquetés *liste* ;
- l'approche *hybride* allie les avantages des deux méthodes précédentes, puisqu'il s'agit de faire du *top-down* et du *bottom-up* à la fois et de prendre l'intersection.

L'opérateur de *suivi de pointeur* (*pointer chasing*) est introduit par [Gardarin *et al.* 1996a] qui adapte les formules de coût par calibration données par [Du *et al.* 1992].

L'opérateur « suivi de pointeur » modélise la traversée des pointeurs à travers un chemin. Pour cela, les paramètres suivant sont introduits :

- PC_0 est le coût initial du suivi de pointeur correspondant au traitement de l'opérateur et la mise en place de la traversée ;
- PC_1 est le coût d'entrées/sorties et le coût CPU pour récupérer un objet par son OID et vérifier le prédicat ;
- PC_2 est le coût de traitement d'un tuple résultat pour le suivi de pointeur ;
- n est le nombre de collections ;
- $fan(C_1, C_2)$ est le nombre moyen de références d'un objet de la classe C_1 à un objet de la classe C_2.

Le coût de traversée de pointeur est défini par :

$$PC = \underbrace{PC_0}_{(0)} + \underbrace{PC_1 \times ||C_1|| \times (1 + \sum_{i=1}^{n-1} \prod_{j=1}^{i} (fan(j, j+1) \times Sel_j))}_{(1)} + \underbrace{PC_2 \times ||C_1|| \times Sel_1 \times Proj \times \prod_{i=1}^{n-1} (fan(i, i+1) \times Sel_{i+1})}_{(2)}$$

où (0) est le coût initial du suivi de pointeur, (1) le coût de chargement des objets en mémoire par leur OID avec évaluation du prédicat et (2) le coût de traitement de l'objet sélectionné par l'opérateur de suivi de pointeur. Le coût de suivi de pointeur se base surtout sur la fonction $fan(C_1, C_2)$ définissant la probabilité de relation entre un objet de la classe C_1 et un objet de la classe C_2. Dans le cas de données semi-structurées, on ne peut pas parler véritablement de classes d'objets, et [McHugh et Widom 1999a] adapte le modèle de coût aux données semi-structurées en définissant les fonctions $Fout_{x,l}$ sur le nombre moyen d'objets sortants étiquetés l reliés aux objets x et $Fin_{x,l}$ sur le nombre moyen d'objets entrants labellisé l reliés aux objets x.

Le facteur de sélectivité est en relationnel assez simple à déterminer si on se base sur des répartitions uniformes. En semi-structuré, il s'agit de considérer le facteur de sélectivité en tenant compte des chemins [Aboulnaga *et al.* 2001]. L'optimisation d'une expression de chemin est similaire au problème d'ordonnancement de jointure en base de données relationnelles, les algorithmes de jointure reposent sur des statistiques sur chaque paire de jointure. Avec des données semi-structurées, il faudrait pouvoir conserver des statistiques sur les chemins complets.

Des formules de coût adaptées à l'architecture du SGBD pour données semi-structurées natifs LORE ont été proposées [McHugh et Widom 1999a]. Pour cela, des informations statistiques à propos de la taille, de la forme et de l'étendue des valeurs doivent être conservées. Initialement, ces statistiques étaient maintenues au niveau du *DataGuide*, mais cela limitait les statistiques aux chemins commençant par un objet nommé, depuis, des index (Lindex, Bindex, Vindex, Pindex) ont été utilisés.

L'approche est de stocker les statistiques sur tous les sous-chemins (séquence d'étiquettes) possibles jusqu'à une longueur k où k est le paramètre de contrôle. Les statistiques conservées pour chaque sous-chemin p de longueur inférieure à k, sont :

- pour chaque type atomique, le nombre total d'objets atomiques de ce type atteignables par p ;
- pour chaque type atomique, les valeurs minimum et maximum de tous les objets atomiques de ce type atteignables par p ;
- le nombre total d'instances du chemin p, noté $|p|$;
- le nombre total d'objets distincts atteignables par p, noté $|p|_d$;
- le nombre total de sous-objets d'arcs entrants étiquetés l de tous les objets distincts atteignables par p, noté $|p_l|$;
- le nombre total de tous les arcs entrants étiquetés l vers chaque instance de p, noté $|p^l|$.

On définit alors :

- $Fout_{x,l} = |p| \times \frac{|p_l|}{|p|_d}$ le nombre d'arcs sortant d'étiquette l associé à x.
- $Fin_{x,l} = |p| \times \frac{|p^l|}{|p|_d}$ le nombre d'arcs entrant d'étiquette l associé à x.

Les formules de coût estiment les coûts d'entrées/sorties, les coûts CPU et le nombre d'évaluations. Lors du choix des plans d'exécution, les coûts CPU ne sont pris en compte que si les coûts d'E/S sont identiques. Tous les opérateurs de l'algèbre LORE ont été détaillés dans [McHugh et Widom 1999a] avec leur coût E/S, CPU et leur nombre d'évaluation. Ainsi, les coûts des différents opérateurs ont été formulés et le tableau ci-dessous nous en donne quelques exemples.

Opérateur	**Coût E/S**	**Coût CPU**	**Nombre d'évaluation**
Projection	$IOCost(fils)$	$CPUCost(fils) + (\|fils\| * CPUCost_{comp})$	$\|fils\|$
Restriction	$IOCost(fils)$	$CPUCost(fils) + (\|fils\| * CPUCost_{eval}(predicat))$	$\|fils\| * selectivite(predicat)$
Scan	$Fout_{PathOf(x),l}$	$\|x\| * CPUCost_{comp}$	$Fout_{PathOf(x),l}$
Pindex	$\|l\| * 2$	$LengthOf(p)$	$\|p\|$
...	...	...	...

5.3.2 Prise en compte des capacités des sources

Les sources peuvent être très variées (SGBD, sources web), et de ce fait, ont des capacités de traitement de requêtes différentes. Si certaines sources comme les SGBD peuvent traiter un grand nombre de type de requêtes, d'autres comme les sources web ont des capacités d'interrogation très limitées. Il s'agit de savoir comment traiter les requêtes appartenant au *domaine de requêtes* valide des sources.

Une première approche est de faire en sorte que chaque adaptateur pallie les déficiences de la source qu'il gère en implémentant les capacités non-gérées par cette dernière. Cette méthode présente l'avantage d'avoir un moyen d'interrogation entre le médiateur et les adaptateurs de capacité uniforme. L'inconvénient est que cela demande à chaque programmeur d'adaptateurs un travail non négligeable. De plus, cela n'est pas toujours possible, par exemple dans le cas de ressources limitées sur la machine supportant l'adaptateur pour pouvoir faire tourner un adaptateur complexe.

Une deuxième approche est de créer un *langage d'exportation* de capacité de la source permettant à celle-ci de spécifier au médiateur quelles sont les requêtes que la source est capable de traiter. C'est au médiateur ensuite de pallier aux manques de la source. La prise en compte de la capacité des sources implique une coopération entre optimiseur et adaptateurs. Afin que le médiateur puisse générer le ou les plans d'exécution en tenant compte de ce que la source il faut que la source puisse communiquer au médiateur ce dont il est capable de faire. [Li *et al.* 1998] [Vassalos et Papakonstantinou 1997] dans TSIMMIS et [Tomasic *et al.* 1996] dans DISCO ont ainsi implémenté un langage de description des capacités de la source. Par exemple, sur DISCO, les capacités de traitement exportées par une source peuvent être :

```
scan (Pub0) ;
select (Pub1 (author (=) year (<))) ;
```

La première déclaration indique que seule la relation `Pub0` peut être lue séquentiellement au moyen de l'opérateur `scan`. La deuxième déclaration signifie qu'il est possible de sélectionner les tuples de `Pub1` dont l'attribut `author` vaut exactement une certaine valeur, et les tuples de `Pub1` dont l'attribut `year` est inférieur à une valeur donnée.

5.3.3 Synthèse

Il existe différentes manières d'exécuter une même requête, nommées aussi plans d'exécution. L'efficacité des plans doit être estimé par un modèle de coût. Ceci permet de choisir le plan celui de coût minimum. Nous avons pour cela étudié différentes méthodes de calculs de coût existants. Beaucoup s'appuient sur des modèles de données relationnelles, objet ou orienté-objet. Très peu de travaux ont été fait sur les données semi-structurées, et nous nous attacherons dans cette thèse à définir un modèle de coût sur un tel type de

données. Les sources pouvant être très hétérogènes, elles peuvent avoir des capacités de traitement de données très différentes, mais aussi avoir des modèles de coût plus ou moins définis. Il s'agit donc de savoir communiquer et intégrer ces différents paramètres. XML est un bon support pour cela.

Lorsque les sources sont centralisées, on connaît les paramètres physiques des sources (CPU, E/S) voire même les formules de coût définies par le constructeur. Dans ce cas le coût des adaptateurs peut être intégré avec précision dans le calcul de coût du médiateur. Si les sources possèdent des comportements réguliers et que l'on connaît les méthodes d'optimisation des requêtes, on peut utiliser la calibration. Dans le cas de sources de comportement très hétérogènes, on peut affiner la méthode de calibration en divisant les catégories de requêtes en classes différentes, chacune ayant son modèle de coût. Il se peut enfin que la source évolue dans un environnement dont les paramètres changent fréquemment, ou que le coût d'exécution de certaines requêtes varient en fonction du temps. Pour cela des méthodes qualitatives et fractionnelles ont été proposées.

La calibration nécessite de la source la possibilité de la calibrer, notamment en autorisant la création et la mise à jour d'une base de calibration (IRO-DB [Gardarin *et al.* 1996b]), ce qui n'est pas toujours possible (sources web). On peut alors recourir à une technique statistique s'appuyant sur l'historique des requêtes effectuées jusqu'alors. La base d'historique est alors crée au niveau du médiateur, ce qui peut le surcharger. HERMES [Subrahmanian *et al.* 1997] permet de modéliser les coûts en n'utilisant que les résultats de requêtes précédentes (basé sur un historique). On interroge lorsque le médiateur est au repos les sources, et on alimente une base locale avec les résultats des requêtes, mais la base locale risque de devenir assez grande. Cette approche n'est efficace que si les requêtes posées ne sont pas trop différentes entre elles.

Enfin, des méthodes alliant l'ensemble d'une partie ou de toute ces possibilités ont été réalisées dans GARLIC et DISCO, permettant à chaque adaptateur d'avoir son propre modèle de coût. GARLIC [Roth *et al.* 1999] énumère tous les plans et interroge les adaptateurs pour déterminer si le plan est faisable et obtenir le coût du plan. Avec son modèle de coût combinant ceux des adaptateurs exportant le leur, et en utilisant un modèle de coût générique pour les autres, DISCO a trouvé une réponse adaptée quand à l'estimation des coûts dans une architecture de médiation. Enfin, [Florescu *et al.* 1999] présente un algorithme permettant d'utiliser les capacités tout en explorant des plans réalisables (optimiseur traditionnel enrichi d'une prise en compte de capacité).

On remarquera néanmoins que les modèles de coût ont été surtout étudiés pour des SGBD relationnels [Du *et al.* 1992] ou objet [Gardarin *et al.* 1996a]. Mais peu d'études ont été réalisées sur un modèle de coût s'appuyant sur des données semi-structurées, [McHugh et Widom 1999a] s'appuie sur l'entrepôt natif semi-structuré LORE pour décrire un modèle de coût basé sur des données semi-structurées. On constate qu'il n'y a pas de travaux sur les modèles de coût basés à la fois sur une architecture de médiation et manipulant des données semi-structurées en interne.

5.4 Paramètre d'un modèle de coût

Les paramètres permettant d'évaluer le modèle de coût d'un système sont constitués de statistiques du système et des données mais aussi de différentes formules.

Nous rajoutons à ce coût générique des coûts adaptés plus spécifiquement aux données semi-structurées. Pour ces besoins, nous définissons un autre langage d'exportation de coût plus générique permettant de déclarer des paramètres et des formules propres à des sources maniant ces nouvelles sources de données. Ce langage de communication d'information de coût sous forme XML a été décrit dans le chapitre 3.

5.4.1 Statistiques

Il y a deux types de statistiques à prendre en compte. Tout d'abord, les statistiques du système : le coût CPU du processeur, les coûts E/S des entrées-sorties, les coûts de communication entre le médiateur et les sources, les coûts d'initialisation de traitement des requêtes, etc. Ces statistiques ne dépendent pas de la requête et des données, et peuvent être connues individuellement ou de façon combinée [Gardarin *et al.* 1996b].

Il y a ensuite les statistiques de données, elles sont relatives aux données et au placement des données dans la base. Il peut s'agir de la cardinalité d'une collection, de la taille des objets d'une collection mais aussi pour les SGBD relationnels ou objets, d'attributs indexés, des valeurs pouvant être prises par l'attribut, du nombre de valeurs distinctes pouvant être prises par l'attribut, etc.

5.4.1.1 Statistiques système

Dans le modèle de coût de type calibration (calibration sur données simples, objets, échantillonnage, qualitatif, fractionnel, probabiliste, générique), certains paramètres dépendant du système (indépendant des données) doivent être déterminés. Ils peuvent être obtenus soit par des informations constructeurs (basées sur le processeur, système d'exploitation, type du SGBD utilisé), soit par calibration.

Les informations constructeurs sont les suivantes :

- **CPU** : temps machine - temps moyen pour exécuter une instruction simple ;
- **ES** : temps moyen d'une entrée/sortie.

On peut modéliser plus précisément le coût de la méthode d'accès par exemple avec la formule de Yao [Yao 1977], mais cela prend en compte des paramètres systèmes aussi précis que la taille d'une page ou la répartition et la fragmentation des données sur le

disque. Ces informations sont rarement accessibles mais dans le cas où elles le seraient, il convient de les prendre en compte et de les utiliser afin de s'approcher le plus possible du modèle réel.

Pour les informations relatives à l'organisation des données dans le SGBD, nous partons de l'approche par calibration de donnée par [Gardarin *et al.* 1996b]. Cette méthode implique les variables systèmes suivantes :

- $\mathbf{SS_0}, \mathbf{SI_0}, \mathbf{SC_0}$: les coûts initiaux du balayage séquentiel, par indexation, par index clusterisé ;
- $\mathbf{SS_1}$: le coût d'entrées/sorties et CPU pour traiter chaque page de la collection (récupération d'objets et vérification des prédicats) dans le cas d'un balayage séquentiel ;
- $\mathbf{SI_1}, \mathbf{SC_1}$: les coûts de consultation d'index respectivement dans le cas d'un balayage séquentiel et dans le cas d'un index clusterisé ;
- $\mathbf{SS_2}, \mathbf{SI_2}, \mathbf{SC_2}$: le coût de traitement des tuples résultats, respectivement dans le cas d'un balayage séquentiel, par index ou par index clusterisé.

Adaptation aux données semi-structurées. En plus de ces variables, pour le cas de données semi-structurées, il faut prendre en compte la notion d'arborescence et de chemins. Ce que la structure arborescente apporte de plus comme opérations sont la notion de comparaison de sous-arbres et de recherche de nœuds par son chemin dans un arbre.

Une comparaison d'arbres de n nœuds chacun se fait en $2*(n-1)$ suivis de références d'arcs, $2n$ chargements de nœuds et n comparaisons des valeurs de nœuds si les deux arbres sont structurés de la même façon. Un suivi d'un chemin de longueur n dans un arbre se fait en un minimum de $(n-1)$ suivis de référence et n chargements de nœuds et en un maximum en $(m-1)^n$ suivis de pointeurs et m^n chargements de nœuds, où m est le nombre moyen de fils par nœuds. En moyenne, il faut effectuer $(\frac{m}{2}-1)^n$ suivis de références et $(\frac{m}{2})^n$ chargements de nœuds. Si le nombre de fils par nœud intermédiaire est spécifié, on peut affiner encore ces résultats.

Parmi les statistiques systèmes on peut déclarer le temps de suivi de pointeurs, le temps de récupération d'un nœud et le temps de comparaison de deux nœuds.

Certains entrepôts de données semi-structurées font appel à des B+Tree et des listes inversées pour indexer les chemins, les valeurs, les étiquettes, etc. Ceci est d'autant de paramètres qu'on peut vouloir considérer dans le calcul du coût (tailles des *buckets*, largeur du B-Tree, etc.).

Nous rajoutons pour prendre en considération les données semi-structurées, les variables systèmes suivantes :

- **COMP_VAL** : le temps de comparaison de deux valeurs ;
- **SUIVI_REF** : le temps de suivi d'un pointeur (passage d'un nœud à un nœud

suivant).

5.4.1.2 Statistiques de données

Les statistiques de données dépendent des données rentrées dans la base.

- **Card**(**C**) : la cardinalité de la collection C (notée aussi $||C||$) ;
- **NDIST**(**a**) : la distribution de l'attribut a ;
- **min**(**a**) la valeur minimum que peut prendre l'attribut a ;
- **max**(**a**) la valeur maximum que peut prendre l'attribut a.

Adaptation aux données semi-structurées Les statistiques des données utilisées en données semi-structurées sont principalement :

- **NB_FILS**(*chemin*, *nom_attribut*) exprime le nombre de nœuds fils d'étiquette *nom_attribut* de parents accessibles par *chemin*.
  Par exemple **NB_FILS**("/*PERSONNE*/*VOITURE*/*COULEUR*") := **1** exprime qu'il y a en moyenne 2 nœuds *COULEUR* par voiture ;
- **profondeur**(*chemin*)
  permet de donner la profondeur du sous-arbre dont la racine est le nœud désignée par le chemin spécifié ;
- **hauteur_arbre**(*nom_collection*)
  hauteur de l'arbre de la collection spécifiée ;
- **largeur_arbre**(*nom_collection*)
  largeur de l'arbre de la collection spécifiée ;
- **NDIST**(*nom_collection*, *nom_attribut*)
  exprime la distribution de l'attribut *nom_attribut* dans la collection ;
- **min**(*nom_collection*, *nom_attribut*)
  donne la borne inférieure du domaine dans lequel varie *nom_attribut* dans la collection ;
- **max**(*nom_collection*, *nom_attribut*)
  donne la borne supérieure du domaine dans lequel varie *nom_attribut* dans la collection ;

5.4.2 Formules de coût

Le modèle de coût d'un système varie en fonction de ses paramètres systèmes et ses paramètres de données (ou de collection). Il s'agit de définir une ou plusieurs formules permettant de calculer le coût d'évaluation d'une requête dans ce système (granularité de calcul la plus grosse) ou d'un prédicat dans un opérateur particulier de ce système

(granularité plus fine).

Une hiérarchie de règles de coût [Naacke 1999] peut être exploitée pour la définition des formules de coût. Le cas le plus précis est un adaptateur donnant sa stratégie d'optimisation, où tous ses paramètres internes et de collections sont connus et où les formules de coût les utilisant sont déclarées. Ce cas là est assez rare. Dans le cas le plus courant sur les SGBD, les adaptateurs peuvent être interrogés sur certaines statistiques de collections (*via* des requêtes adéquates), et calibrés [Gardarin *et al.* 1996b] pour déterminer certains des paramètres internes. Avec ces données, on utilise ensuite des formules générales associées à chacun des opérateurs (restriction, projection, jointure, etc.). Enfin, dans le cas de sources vraiment opaques, il est possible d'utiliser un modèle de coût générique [Roth *et al.* 1999]. On détermine par des tests adaptés le temps moyen de récupération d'un premier tuple (comportant ainsi la phase d'initialisation) appelé aussi *temps à froid*, ainsi que le temps de récupération d'un tuple suivant (*temps à chaud*). Le temps total s'approxime par :

$$T_{total} = T_{froid} + (card - 1)T_{chaud}$$

- `TF` : temps à froid - temps pour récupérer le premier tuple résultat ;
- `TC` : temps à chaud - temps pour récupérer un tuple résultat hormis le premier.

Cette formule générique réside par défaut au niveau du médiateur et est adaptable à un adaptateur quelconque ne donnant aucune information de coût.

Pour tous les cas où l'adaptateur a des informations de coût à communiquer, le langage de communication de coût défini dans le chapitre 3 peut être utilisé.

5.5 Intégration de modèle de coûts des adaptateurs

À partir des règles de grammaire définies ci-dessus, un adaptateur peut définir les statistiques et les formules de coût qu'il veut exporter. Ces formules et paramètres seront pris en compte par l'évaluateur de coût du médiateur accédant à l'adaptateur.

5.5.1 Modèle de coût d'une source native de données semi-structurées

Afin d'étudier le comportement d'un entrepôt natif de données, nous allons exposer rapidement l'architecture du composant ReposiX développé au sein du laboratoire PRiSM pour le projet MUSE.

La figure 5.1 décrit l'architecture de ReposiX, c'est un système autonome permet-

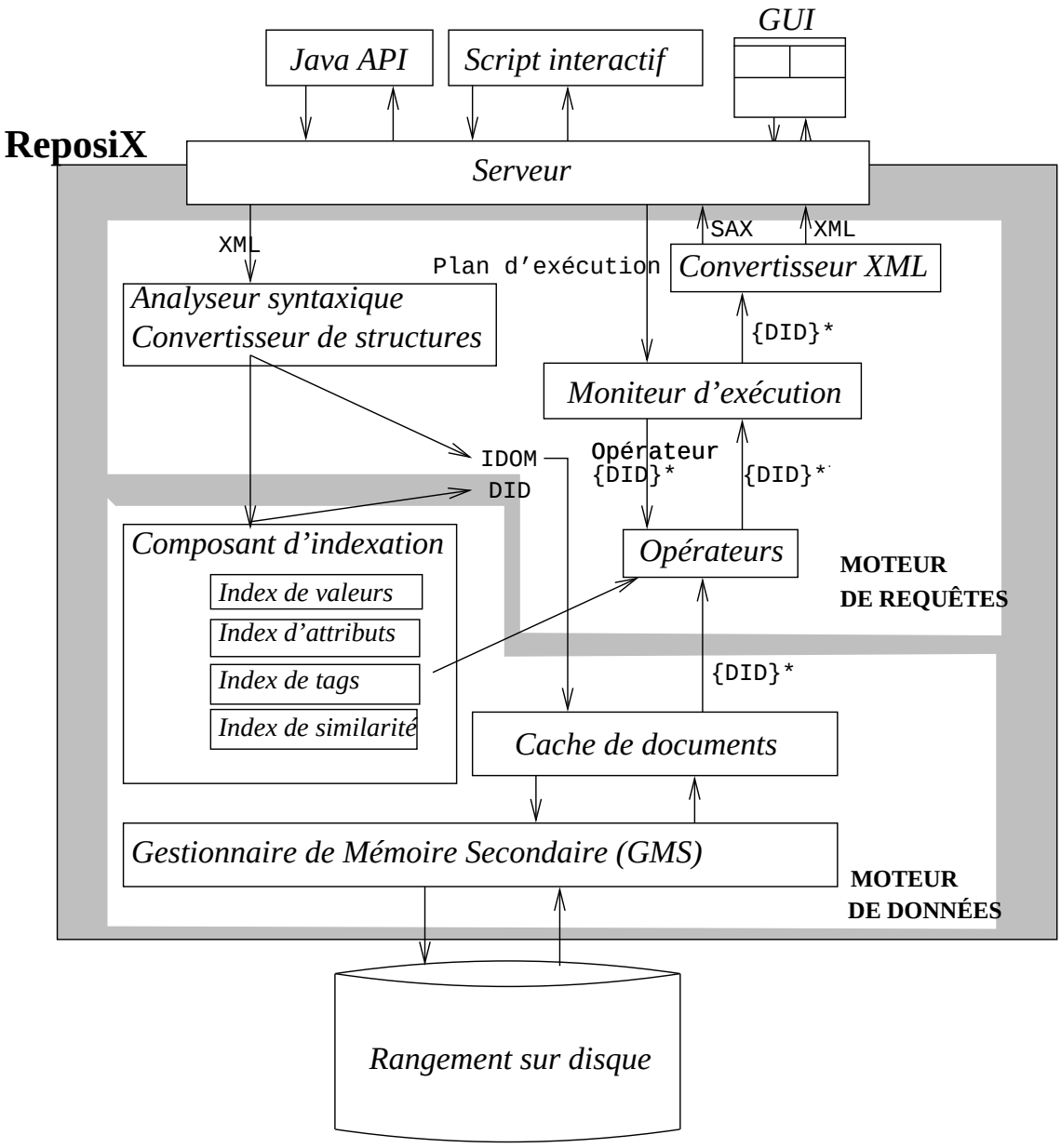


FIG. 5.1 – Architecture de ReposiX

tant de stocker et de récupérer des documents XML suivant un format compact sérialisé spécifique. Son interrogation est très simple et se fait *via* un langage à base d'opérateurs. Un moteur de requête permet d'interfacer les opérations entre les programmes utilisateur et la couche de stockage logique. Le moteur de données ou couche de stockage logique s'occupe de la sérialisation des documents et des diverses indexations (valeurs, chemins, etc.) de ceux-ci. Enfin la dernière couche est la couche physique représentée par le disque.

Le système ReposiX comporte un index de chemins basé sur un classique B-Tree

avec index inversé [Cutting et Pedersen 1990]. Ainsi, par l'intermédiaire de ces structures, pour un chemin donné, on peut récupérer l'ensemble des emplacements comportant ces nœuds. Les nœuds sont stockés sous forme sérialisée suivant un codage spécifique similaire à PDOM [Huck *et al.* 1999].

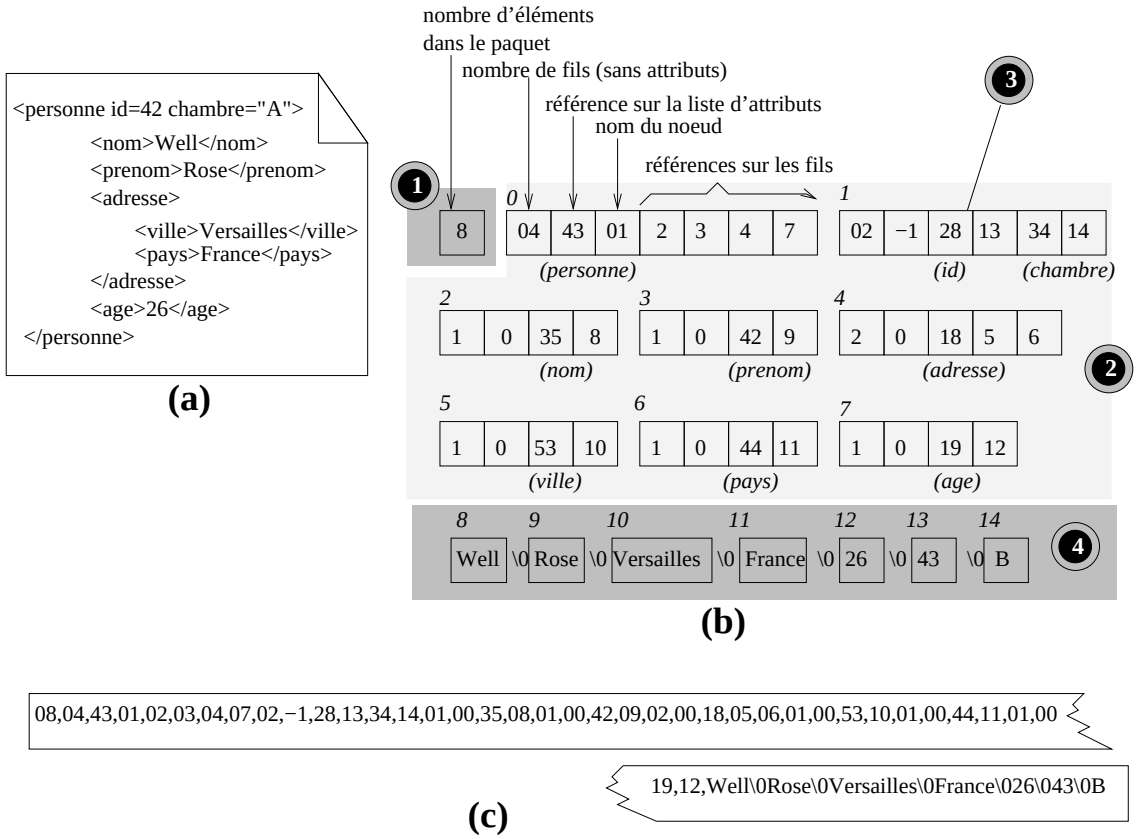


FIG. 5.2 – Sérialisation d'un document XML sous ReposiX

La figure 5.2 montre comment un document XML (a) est décomposé de la façon représentée en (b) : la première partie ① est un entier codant le nombre de groupes, la seconde partie ② est la liste des éléments, et la troisième partie ④ est la liste des valeurs. Tous les éléments et attributs sont référencés par des références locales débutant par 0 jusqu'au nombre de valeurs (codés dans la première partie).

Dans la liste des groupes, il y a des éléments nœuds et des groupes d'attributs sérialisés de la façon suivante : si c'est un nœud (tous les groupes de ② sauf le groupe ③), alors le nombre d'éléments est stocké, et la référence locale sur un objet de type « groupe d'attributs » (0 s'il n'y a aucun attribut), puis vient un entier codant le nom du nœud dans le dictionnaire, enfin vient les références locales sur les éléments. Si c'est un groupe d'attributs (groupe ③), la première case est le nombre d'attributs référencés dans le groupe, puis vient -1 (ce qui différencie d'un élément), et finalement des couples $nom\_attribut, valeur$ où $nom\_attribut$ est un entier codant le nom de l'attribut dans le

dictionnaire d'attributs et *valeur* l'entier codant la valeur dans le dictionnaire de valeurs.

Dans la partie des valeurs, toutes les valeurs sont reportées comme des entiers codés dans le dictionnaire des valeurs.

Ainsi, une fois un nœud localisé sur le disque, le sous-arbre correspondant peut-être extrait en autant d'entrées/sorties que le nombre de pages disque correspondant à la taille du document (en prenant pour taille de page disque une valeur de 4ko, on peut raisonnablement estimer qu'un sous-arbre s'extrait en une seule E/S).

L'avantage de cette structure pour représenter un arbre est que les éléments sont déjà indexés, et donc l'accès à un élément se fait en un seul accès par pointeur.

Une récupération d'un nœud se fait en

$$\begin{aligned} R_{E/S} =& Cout\_recuperation\_noeud(pos\_disque) \\ =& \lceil \frac{taille\_de\_arbre\_serialise}{taille\_page} \rceil \ \mathbf{E}/S \end{aligned}$$

et consomme un temps de R_{CPU} correspondant à la désérialisation de l'arbre associé au nœud.

Une récupération d'un nœud par son chemin consiste à identifier le chemin dans le B+Tree avec sa liste inversée, soit :

$$\begin{aligned} B_{E/S} =& Cout\_recherche\_position\_noeud(chemin) \\ B_{E/S} =& \log_b N - \log_b C \ \mathbf{E}/S, \ pour N \geq C \end{aligned}$$

où b est le nombre d'entrées moyen dans une page, N le nombre d'entrées dans le B-Tree, C est nombre d'entrées stockées dans le cache. $\log_b N$ est la profondeur du B+Tree. Soit B_{CPU} le temps CPU d'identification d'un chemin dans le B+Tree avec sa liste inversée.

et

$$\begin{aligned} Cout\_recuperation\_noeud(chemin) =& Cout\_recherche\_position\_noeud(chemin) \\ &+ Cout\_recuperation\_noeud(pos\_disque) \\ =& \log_b N - \log_b C + \lceil \frac{taille\_de\_arbre\_serialise}{taille\_page} \rceil \ \mathbf{E}/S \end{aligned}$$

Une fois un nœud repéré et chargé en mémoire, on peut effectuer des opérations de l'algèbre relationnelle (sélection, projection, jointure, union, intersection, etc.) Suivant la nature même de la structure de l'arbre sérialisé, l'accès aux différents nœuds internes et

aux valeurs est alors très rapide car il ne s'agit plus que des sauts de pointeurs suivant des positions locales très déterminées.

Une fois le nœud récupéré, une restriction ne nécessite plus d'entrées/sorties, elle se limite à identifier les éléments utiles de l'arbre sérialisée à réécrire le nœud suivant cette forme. Une projection se limite à repérer les valeurs utiles à la vérification du prédicat, pour déterminer si l'élément est valide ou non. Un produit cartésien consiste à réunir deux à deux les nœuds, et donc récupérer $n \times m \times Cout\_recuperation\_noeud(chemin)$. Une jointure dans cette architecture consiste en un produit cartésien suivi d'une restriction.

Opérateur	**Coût E/S**	**Coût CPU**
projection	$B_{E/S} + n * R_{E/S}$	$B_{CPU} + n * R_{CPU}$
restriction	$B_{E/S} + n * R_{E/S}$	$B_{CPU} + n * (R_{CPU} + Comp)$
produit cartésien	$B_{1E/S} + B_{2E/S} + n1 * R_{1E/S} + n2 * R_{2E/S}$	$B_{1CPU} + B_{2CPU} + n1 * R_{1CPU} + n2 * R_{2CPU} + n1 * n2 * Comp$

Le tableau ci-dessus résume les formules de coût pour les opérateurs relationnels classiques. À ceci se rajoute des opérateurs sur les ensembles (intersection, union) fréquemment utilisés dans les évaluations de requêtes de ReposiX.

5.5.2 Modèle de coût d'un SGBD-R simple

Nous étudions dans cette sous-section, le cas d'une source de données relationnelles classique. Nous considérons pour cela les formules de coût bien connus du monde relationnel pour chacun des opérateurs suivants :

Projection	$C(\pi_{att} R) = CPU \times \|\|R\|\| + ES \times \|\|R\|\| \times \|att\|$
Restriction	$C(\sigma_{pred} R) = s \times CPU \times \|\|R\|\| + ES$
Jointure (par boucle imbriquée)	$C(R_1 \bowtie_{\sigma_{att=x} R_2} = \|\|R_1\|\| \times C(\sigma_{att2=x} R_2)$ $+COM \times \|\|R_1\|\| \times \|\|R_2\|\|$
Jointure (par hachage)	$C(R_1 \bowtie_{pred} R_2) = CPU \times \|\|R_1\|\| \times \|\|R_2\|\|$

Un adaptateur gérant un base de données ayant un tel modèle de coût exporterait ces informations par le langage d'exportation de coût que nous avons défini dans le chapitre 3. Le fichier d'information de coût exporté aurait la forme suivante :

```
<costmodel>
  <statistics>
    <system>
    <!-- déclaration de la valeur de CPU -->
  <declare type="real"> <ci>CPU</ci> <cn>0.1</cn> </declare>

  <!-- déclaration de la valeur de ES -->
  <declare type="real"> <ci>ES</ci> <cn>0.2</cn> </declare>
    </system>

    <collection>
  <reln>
    <eq> <apply> <ci>Card</ci> <cn>LIVRE</ci> </apply> <cn>327</cn> </eq>
  </reln>
  <reln>
    <eq> <apply> <ci>Card</ci> <cn>PERSONNE</ci> </apply> <cn>8535</cn> </eq>
  </reln>
    </collection>
    [...]
  </statistics>
<formulas>

  <operators>
  <!-- declaration de la fonction de projection -->
<declare type="fn"> <ci>projection></ci> <lambda>
<bvar> <ci>R</ci> <ci>att</ci> </bvar>
<apply>
<plus>
  <apply>
  <times> <ci>CPU</ci> <apply> <ci>Card</ci> <ci>R</ci> </apply> </times>
  </apply>

   <apply>
   <times> <ci>ES</ci> <apply> <ci>Card</ci> <ci>R</ci> </apply> <ci>att</ci>
   </times>
   </apply>
</plus>
</apply>
</lambda> </declare>
  </operators>

  <!-- declaration des autres fonctions -->
  [...]
  </formulas>
</costmodel>
```

Ce fichier est divisé en une partie déclaration des statistiques systèmes et de collection, et une partie formules d'opérateurs.

5.6 Intégration du modèle de coût dans le médiateur

Dans le chapitre précédent sur l'algèbre utilisée dans notre architecture, nous avons montré que le modèle de données que nous avons proposé permettait de s'affranchir plus ou

moins de la notion de chemin pendant l'évaluation, et d'utiliser l'efficacité des évaluations de l'algèbre relationnelle.

L'algorithme permettant de calculer le modèle de coût au niveau du médiateur se décrit par :

```
fonction coût (XOperateur xop)

    si xop est un XSource
     coût_calculé := coût_traitement_local (cardinalité) + coût_source (xop)
      retourner (coût)
    fin si

    // sinon, si xop est un autre XOperateur que XSource
   initialiser_tableau (tab_coût_entrée)
    pour chaque entrée de noeud
       ajouter_tableau (tab_coût_entrée, coût (entrée))
    fin_pour
   coût_calculé := coût_traitement_local (cardinalité) + évaluer (tab_coût_entrée)
    retourner (coût_calculé)
fin_fonction
```

Le coût d'un XOpérateur se calcule en fonction du coût de traitement de chacun des XOpérateurs dont son ou ses entrées dépendent, ainsi que de son propre coût de traitement.

L'opérateur XSource n'a pas de XOpérateur en entrée. Il doit par contre tenir compte du coût de la source distante, de la communication des résultats et de leur transformation en XTuple.

Le coût des autres XOpérateurs, qu'ils soient unaires, binaires ou n-aires est calculé en fonction des XOpérateurs immédiats dont son ou ses entrées dépendent. En fonction des propriétés de parallélisation du XOpérateur, les flux d'entrées peuvent être sérialisés ou parallélisés. Le coût de traitement varie alors en fonction du degré de parallélisme.

Ainsi, pour un XOpérateur dont le degré de parallélisme est égal à zéro :

$$Cout_{XOp\_serialise}(fils_0, ..., fils_n) = \sum_{i=0}^{n}(Cout_{fils_i}) + Cout_{Op}(card(fils_0), ..., card(fils_n))$$

Et pour un XOpérateur dont le degré de parallélisme est maximal (égal à 1) :

$$Cout_{XOp\_parallele}(fils_0, ..., fils_n) = max_{i=0}^{n}(Cout_{fils_i}) + Cout_{Op}(card(fils_0), ..., card(fils_n))$$

Les algorithmes des différents opérateurs sont détaillés dans la section 4.6 du chapitre 4.

5.6.1 Intégration du coût des adaptateurs : XSource

Pour tous les XOpérateurs de type XSource, le coût est en partie constitué du coût de la requête sur l'adaptateur associé (coût sur la source). Le coût de la source est communiqué par les informations de coût exportées par la source concernée (cf. section 5.4). À cela peuvent s'ajouter les coûts de communication, calculé en fonction du débit du réseau et de la taille et de la cardinalité des résultats renvoyés, S'ajoute à cela le coût du traitement des résultats (transformation en XTuple).

On peut résumer ces considérations en la formule suivante :

$$\boxed{Cout_{XSource}(adapt)) = Cout(adapt) + Cout\_comm(card) + Cout_{XSource}(card)}$$

La cardinalité et le coût de soumission d'une requête Q à l'adaptateur est calculé suivant les informations de coût qu'aura envoyé l'adaptateur de la source lors de l'initialisation.

$$Cout(Adapt(Q)) = Cout_{adaptateur}(Q)$$
$$Card(Adapt(Q)) = Card_{adaptateur}(Q)$$

Le coût de communication entre le médiateur et un adaptateur pour une requête Q est constitué du coût de communication de la requête proprement dite, ainsi que du coût de communication des résultats. Le coût de communication du résultat est fonction du débit du réseau reliant le médiateur à l'adaptateur, de la cardinalité du résultat et de la taille d'une réponse. Ces informations sont elles-aussi données par les informations de coût de l'adaptateur ou du médiateur.

$$\boxed{Cout\_comm(Adapt(Q)) = Taille(Q) * COM + Card(Q) * Taille(reponse) * COM}$$

Finalement le coût de traitement local de l'opérateur XSource se traduit par la transformation des documents résultats en XTuple. Pour une collection de $Card(Q)$ arbres résultats de n nœuds et m feuilles, il y a $Card(Q) \times (n+m)$ éléments à traiter. À chacun de ces nœuds est lié les deux évènements SAX `beginElement` et `endElement`, avec en plus pour les nœuds feuilles, l'évènement `text ()`. Ce qui fait par XTuple :
$E = (\underbrace{2}_{document} + \underbrace{(n \times 2)}_{noeuds\ intermediaires} + \underbrace{m \times (2+1)}_{feuilles})$ évènements à traiter.

Si l'on considère x XAttributs, on a au minimum x (références sur le même chemin) et au maximum $(x+n)$ (références uniquement sur des feuilles, et profondeur maximale) nœuds à matérialiser par XTuple.

Soit CPU_{evt} le temps le traitement d'un évènement SAX, et CPU_{noeud} le temps de matérialisation d'un nœud. On a finalement le temps de traitement local de l'opérateur source :

$$\boxed{Cout_{XSource}(Card(Q)) = Card(Q) \times (E + moy(x, (x + n)))}$$

5.6.2 Coût des XOpérateurs

Pour les XOpérateurs non-sources, le coût est basé sur la cardinalité des XOpérateurs d'entrée et sur le coût de l'opérateur concerné.

Suivant le type de l'opérateur Op, le coût $Cout_{Op}$ et la cardinalité $Card\_Op$ se calculent différemment.

Du fait de la structure interne tabulaire que nous avons décrite dans le chapitre précédent, les coûts présentés se calculent de façon très semblables aux coûts de l'algèbre relationnelle.

Nous avons vu lorsque nous avons décrit les différents XOpérateurs, que l'évaluation se décomposait en une phase d'initialisation et une phase d'exécution.

La phase d'initialisation consiste à repérer avant que les flux d'entrées soient présents les opérations à exécuter : contenance des chemins, nœuds à supprimer ou à conserver, référencement, etc. Cette phase est coûteuse mais est faite qu'une seule fois pour toute. Dans une extension future du médiateur, cette phase pourrait être compilée préalablement de sorte à pouvoir manier des requêtes compilées qui pourront être exécutées ensuite sans cette phase de préparation. Dans notre implémentation, cette phase est effectuée lors de la récupération du premier XTuple. On parle dans ce cas là de l'évaluation « à froid » du XTuple.

Son coût s'écrit :

$$\boxed{cout\_XOperateur(afroid) = initialisation + cout\_op\_relationnel\_init + cout\_op\_arbre}$$

La phase d'exécution se fait lors du traitement des XTuples suivants (ou dans le cadre d'une requête pré-compilée lors du traitement de tous les XTuples). Dans ce cadre là, le traitement se fait « à chaud ».

Les calculs de cardinalité sont exactement les mêmes que dans le cadre relationnel.

5.6.2.1 Projection

La projection s'effectue en supprimant les colonnes non-utiles de la partie XAttribut. Les coûts sont donc les mêmes que pour l'algèbre relationnelle.

Cardinalité Lors d'une projection avec doublons, pour les n XTuples en entrée, n sont retrouvés en sortie.

$$\boxed{Card(\Pi_x(R)) = Card(R)}$$

Coût d'initialisation L'initialisation se fait en calculant pour chaque XAttribut, les inclusions de chemins par rapport à d'autres XAttributs afin de marquer les nœuds des arbres à détruire. Pour une relation de x Xattributs, le coût d'initialisation est de :

$$\boxed{Cout(\Pi_x(R,a)) = x^2 \times CPU_{comp\_chemin}}$$

Coût d'exécution à chaud

$$\boxed{Cout_{chaud}(\Pi_x(R,a)) = (x \times CPU_{report}) + (d \times CPU_{destruction\_noeud})}$$

5.6.2.2 Restriction

Le coût de la restriction est similaire à celui de la restriction relationnelle puisque le comportement des XAttributs est le même. Le surcoût réside essentiellement dans les opérateurs de comparaison qui peuvent faire intervenir des comparaisons de sous-arbres. Mais on peut supposer que dans la majorité des cas, lorsque l'on a à comparer des attributs, cela se fait au niveau des feuilles, et donc que le niveau du sous-arbre à comparer est de 1.

Cardinalité

$$Card(\sigma(pred, R)) = s * Card(R)$$

avec la sélectivité s valant suivant le type du prédicat :
$s(a = valeur) = \frac{1}{NDIST(a)}$
$s(a > valeur) = \frac{max(a) - valeur}{max(a) - min(a)}$

$s(a < valeur) = \frac{valeur - min(a)}{max(a) - min(a)}$
$s(a\ dans\ liste\_valeur) = (\frac{1}{NDIST(a)}) * Card(liste_{valeur})$
$s(P \wedge Q) = s(P) * s(Q)$
$s(P \vee Q) = s(P) + s(Q) - (s(P \wedge Q))$
$s(\neg P) = 1 - s(P)$

Coût d'initialisation Il n'y a pas d'initialisation.

$$\boxed{Cout(\sigma(pred, R) = 0}$$

Coût d'exécution à chaud Dans le cas des opérateurs classiques de comparaison $=, <, >, \leq, geq, \neq$, le coût à chaud dépend du temps de comparaison.

$$\boxed{Cout_{chaud}(\sigma(pred_{\neq'=='}, R)) = CPU_{comp\_val}}$$

Le cas de l'opérateur == est particulier en XQuery et joue sur la restriction non pas d'un attribut mais de tout le sous-arbre associé à l'élément.

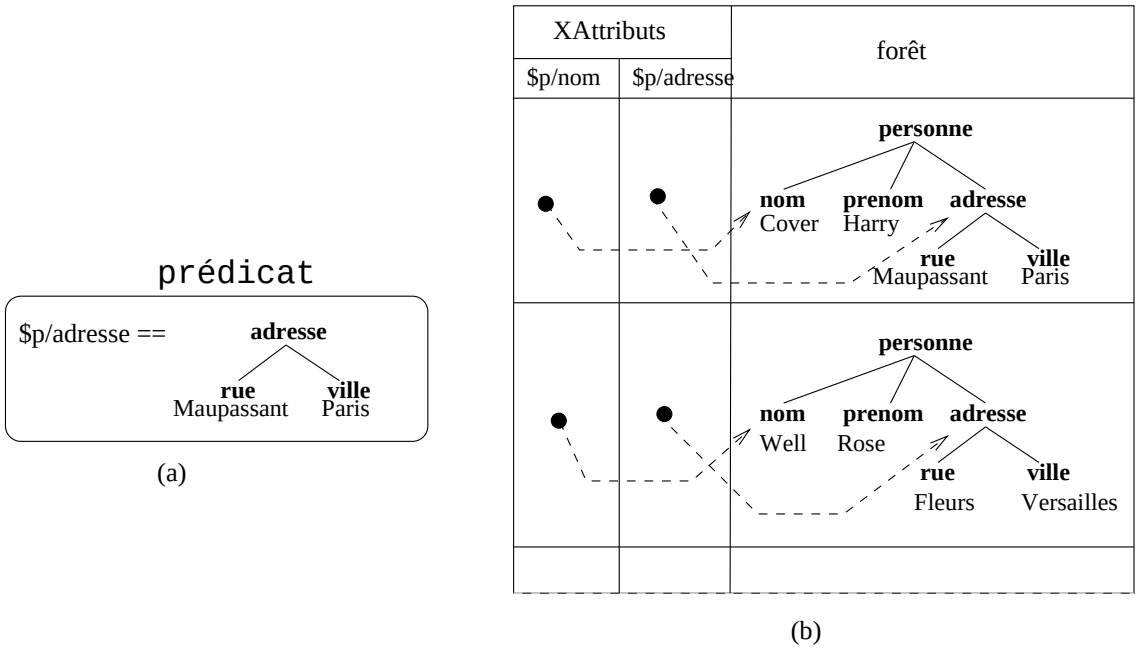


FIG. 5.3 – Cas de comparaison d'attributs de type arborescent

Par exemple, la figure 5.3 montre un prédicat (a) appliqué à un ensemble de résultats intermédiaires de XTuple (b). Le modèle de coût doit donc prendre en charge le coût de

comparaison d'arbre. Pour deux arbres de m fils moyen par nœuds et de n niveaux, donc de $N = \sum_{i=0}^{i=n} m^n = \frac{m^{n+1}-1}{m-1}$ nœuds et de $E = \sum_{i=1}^{i=n} m^n = N - 1$ étiquettes, le coût de comparaison de deux arbres est de :

$$\boxed{N * CPU_{comp\_val} + E * CPU_{ref} = N * CPU_{comp\_val} + (N-1)CPU_{ref}}$$

Et donc finalement :

$$\boxed{Cout(\sigma(pred, R)) = x_2 * (N * CPU_{comp\_val} + (N-1)CPU_{ref})}$$

5.6.2.3 Jointure

Cardinalité

$$\boxed{Card(R1 \bowtie R2) = s * Card(R1) * Card(R2)}$$

avec

$s = 0$ si aucun tuple ne joint
$s = \frac{1}{NDIST(a)}$
$s = 1$ si produit cartesien

Coût d'initialisation Le coût d'initialisation est calculé suivant le coût de fusion présent dans l'algorithme de jointure. L'algorithme de fusion nécessite le marquage des x XAttributs déduit par la comparaison des XAttributs entre eux.

$$\boxed{Cout_{initialisation}(R1 \bowtie R2) = x^2 \times CPU_{comp\_chemin}}$$

Coût d'exécution à chaud Le plus long sous-chemin commun entre deux Xattributs se fait de manière simple (comparaison de deux séquences de noms d'arcs). Le calcul est fait une seule fois et est ensuite valable pour tous les tuples.

De la même façon, chercher le plus long préfixe commun entre un chemin et un ensemble d'autres chemins, se fait simplement et une seule fois.

Par contre pour chaque tuple, dans le cas où les arbres utilisés sont communs, on peut noter un parcours de chemin dans un arbre où le nombre de nœuds parcouru est compris entre 1 et $max(chemin_1, chemin_2)$.

$$\begin{aligned} \textit{Par boucle imbriquee} \quad & Cout(R1 \bowtie R2) = Card(R1) * Card(R2) \\ \textit{Par hachage} \quad & Cout(R1 \bowtie R2) = Card(R1) * (1 + Card(R2)) \end{aligned}$$

5.6.3 Coût de la reconstruction

Le coût du module de recomposition est fonction du nombre total d'objets dans la XRelation finale de n XTuples de x XAttributs chacun.

Soit

$$\boxed{Cout(R) = Cout_{construct} \times x \times n}$$

5.7 Conclusion

Le modèle de coût de l'architecture de médiation s'appuie sur un modèle de coût générique tel que défini par DISCO que l'on a étendu afin d'intégrer complètement les données semi-structurées. Ainsi, le langage d'exportation de coût permet à présent l'exportation de statistiques basées sur des manipulations propres aux données semi-structurées (recherche de chemins). Les formules de coût de l'algèbre au niveau médiateur ont aussi été étendues afin d'intégrer ce nouveau type de données. Enfin, nous avons étudié le modèle de coût d'une source basée sur du semi-structurée natif et nous avons les formules de coûts des opérateurs algébriques traditionnels aux opérateurs étendus que nous avons étudiés dans le chapitre précédent. L'intégration des données d'autres types de sources (classiques : relationnelles, objets ou spécifique : source web) n'a pas été décrite, car elle s'effectue de la même façon que dans le modèle de coût générique : on utilise les formules et coût exportées par les sources si elles sont données, et on utilise un modèle par défaut dans les cas contraire.

Nous avons décrit le modèle de coût associé à chaque opérateur du médiateur. L'exécution d'un opérateur algébrique se décompose en deux phases : ① une phase d'initialisation permettant de définir les opérations à exécuter lors de la phase d'exécution. ② une phase d'exécution appliquant les opérations préparées par la phase d'initialisation à chacune des données des flux d'entrée. Les calculs de cardinalité sont les mêmes que ceux de l'algèbre relationnel. La phase d'initialisation est spécifique à notre algèbre et coûteuse puisqu'il s'agit de préparer l'ensemble des opérations à effectuer. Les modèles de coût pour cette phase sont spécifiques à notre modèle. Dûe à la structure tabulaire de l'algèbre que nous avons définie, les modèles de coût de la phase d'exécution diffèrent peu de ceux de l'algèbre relationnelle.

Chapitre 6

Cache sémantique pour médiateur de données semi-structurées

6.1 Introduction

Les sources de données fédérées par une architecture de médiation peuvent être dispersées sur un vaste réseau comme l'Internet dont le temps de réponse par rapport à une base de données locale peut être important. Il est donc avantageux d'utiliser un composant local - nommé *cache* [Franklin *et al.* 1993] - permettant de stocker temporairement les données envoyées par les sources en réponse à des requêtes, de sorte à limiter le trafic si d'autres requêtes accédant à ces mêmes données devaient être formulées par la suite.

Il s'agit dans un premier temps de définir la façon dont seront stockées les données récupérées par les sources. Comme nous travaillons dans un contexte semi-structuré, nous nous appliquerons à utiliser comme cache un entrepôt de données gérant des données semi-structurées. Nous montrerons ensuite comment une telle architecture de cache peut s'intégrer dans notre architecture. Il s'agit ensuite de déterminer les critères utilisés pour stocker les données dans le cache et - la taille du cache n'étant pas infinie - quelle est la politique de mise à jour de ce cache. Les sources de données étant très hétérogènes et réparties, nous ne pouvons pas nous appuyer sur des mécanismes de réplication de pages ou d'identifiants (les sources ne diffusant pas toujours leur politique interne de stockage). Nous nous basons seulement sur les réponses des sources à des requêtes déjà posée. Pour cela, nous nous appuyons sur la sémantique des requêtes [Adali *et al.* 1996]. Nous verrons dans ce chapitre comment la sémantique des requêtes est utilisée pour déterminer quelles sont les données déjà stockées dans le cache, en tenant compte spécialement des données semi-structurées. Enfin, nous verrons comment intégrer ce cache dans le calcul des modèles de coût que nous avons défini dans le chapitre précédent.

6.2 Plan du chapitre

Nous présentons tout d'abord un aperçu sur les caches dans les architectures de médiation dans la section 6.3. Nous développons ensuite un aperçu sur le stockage de données XML dans la section 6.4. Nous montrons dans la section 6.5 comment utiliser un entrepôt de données XML comme cache. Nous décrivons la gestion d'un tel cache au sein d'une architecture de médiation dans la section 6.6. Nous indiquons comment intégrer le cache au modèle de coût dans la section 6.7. Et enfin nous concluons section 6.8.

6.3 Techniques de gestion de caches

Dans une architecture client/serveur, dû au coût de communication, il peut s'avérer utile de s'interroger sur la réutilisabilité des résultats de requêtes précédemment exprimées.

Si au niveau des sources elles-mêmes, il est fréquent qu'il y ait une optimisation des entrées/sorties disque basée sur un cache des blocs disques ou pages fréquemment accédés, cela ne peut s'effectuer de cette manière au niveau du composant de médiation. Dans les architectures client/serveur standards, l'unité de transfert entre les serveurs et les clients sont les *pages* ou les *tuples*. Dans une architecture XML, il s'agit de fragments XML, ce qui nécessite de définir des stratégies de gestion de cache nouvelles.

6.3.1 Basé sur les pages

Les mécanismes à base de *cache de pages* sont largement utilisés dans les SGBD, ils prennent pour hypothèse que chaque requête posée au client peut être traitée localement et décomposée au niveau requête sur des pages individuelles. De cette façon, si une page demandée n'est pas présente dans le cache client, une requête sur la page complète est envoyée au serveur.

Une telle méthode n'est pas appropriée à un système de fédération de données semi-structurées. En effet du fait de la structure de graphe des données semi-structurées, le découpage physique de la structure peut se faire de multiple façons sur les pages disques du serveur et ne pas être appropriée et prédictible par le client. De plus, les documents XML s'appuyant par exemple sur des serveurs Web, ne sont souvent interrogeables que de façon limitée (mot-clef), et donc le transfert par page est impossible. Enfin, la diversité des organisations des différentes sources ne rend pas homogène la façon d'organiser les pages.

6.3.2 Basé sur les tuples

Dans un mécanisme de cache par tuple, le cache est maintenu sous forme de tuples individuels, permettant un plus haut niveau de flexibilité que le cache par page.

Sur le Web, le cache de tuples est faisable car les documents web peuvent être référencés et accédés par leur URL (*Uniform Resource Locator*). Dans un *proxy* c'est ainsi que un cache des pages web récemment accédées sont conservés.

Mais, il est souvent difficile de faire une requête à la source en lui précisant quels sont les tuples qui existent déjà dans le cache et donc de réduire la taille de la réponse. De la même façon, les clients peuvent difficilement détecter si leur cache local comporte une réponse complète à la requête. Du coup, les clients sont forcés d'ignorer les tuples dans le cache en effectuant leur requête. Ce qui a pour conséquence que lorsque les clients reçoivent les réponses, ils doivent ensuite détecter les duplicata avec leur propre cache.

6.3.3 À base de prédicats ou cache sémantique

Pour pallier aux inconvénients des caches par page et par tuple dans le cas d'une médiation hétérogène de données semi-structurées, les caches sémantiques ont été proposés [Dar *et al.* 1996]. Dans l'approche de [Keller et Basu 1996], les requêtes sont exécutées sur le serveur et utilisées pour charger le cache client. La description des prédicats utilisés est stockée au niveau du client et au niveau du serveur. Ainsi, pour une nouvelle requête, un client examinera s'il peut l'exécuter localement, et dans le cas contraire, enverra toute ou une partie de la requête au serveur pour exécution, puis mettra son cache à jour. Le cache pouvant avoir ses propres index et chemins d'accès afin de faciliter les exécutions de requêtes locales. L'unité de transfert est ici une région sémantique. Lorsqu'une requête est posée à un client avec un cache sémantique, la requête est divisée en deux parties :

1. Une requête locale qui récupère la partie des réponses disponibles dans le cache local.
2. Une requête complémentaire permettant de récupérer les données manquantes sur le serveur. Si la requête complémentaire est non nulle, elle est envoyée vers le serveur et y est exécutée.

L'article de [Chidlovskii *et al.* 1999] présente les différents cas possibles entre une requête utilisateur Q et des régions du cache. La figure 6.1 illustre ces cas. Les cas sont les suivants :

1. *l'équivalence* : le cache contient une région R dont la formule est équivalente à la requête Q ;
2. *l'inclusion de la requête* : le cache contient une ou plusieurs régions contenant la requête Q ;

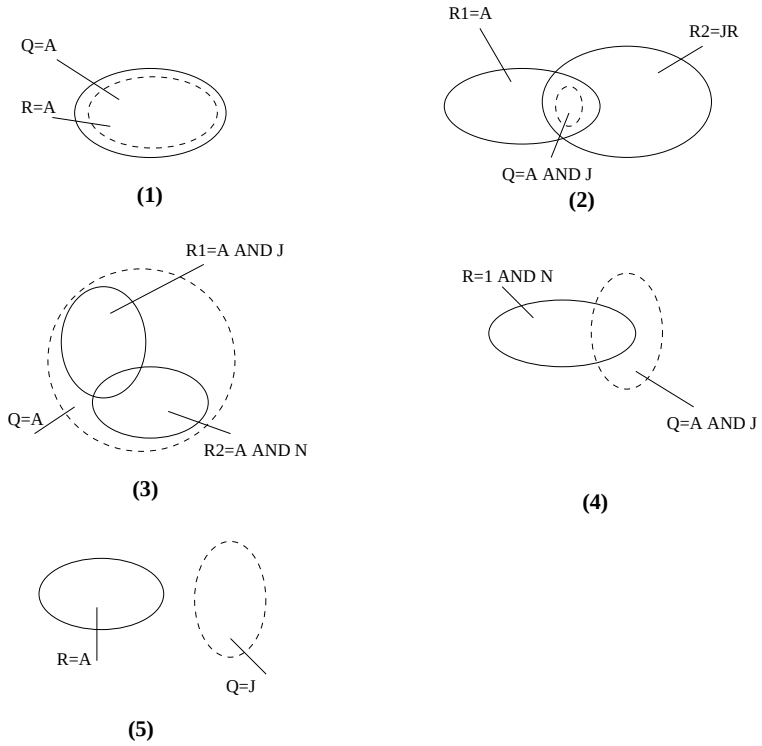


FIG. 6.1 – Les différents cas possibles pour un cache

3. *l'inclusion des régions* : le cache contient des régions du cache dont les formules sont contenues dans la formule de la requête ;
4. *l'intersection* : il y a intersection entre les régions du cache et la requête ;
5. *la disjonction* : les régions du cache et la requête sont disjointes.

Ces cas se traitent comme présentés dans le tableau suivant :

	(1)	(2)	(3)	(4)	(5)
Formule	$Q \equiv R$	$Q \subset R_i$	$R_i \subset Q$	$R_i \bigcap Q \neq \emptyset$	$R_i \bigcap Q = \emptyset$
Traité	R	$Q \bigcap R$	$\bigcup R_i$	$Q \bigcap (\bigcup_i R_i)$	$\emptyset$
Reste	$\emptyset$	$\emptyset$	$Q - \bigcup_i R_i$	$Q - \bigcup_i d_i$	Q
Région du cache	pas de changement	pas de changement	ajouter Q à R_i	nouvelle région $Q - \bigcup_i d_i$	nouvelle région Q

Les colonnes représentent les cas possibles de recouvrement entre une requête utilisateur et une région du cache ainsi que nous l'avons énuméré précédemment. Les lignes traitent ensuite de :

1. *formule* : la définition du cas de façon formelle ;
2. *traité* : ce qui peut être traité par le cache ;

3. *reste* : ce qui n'est pas traité par le cache et qui reste à traiter ;
4. *région du cache* : les changements à apporter à la région du cache concernée.

Dans les cas (1) et (2), les résultats de la requête utilisateur sont entièrement contenus dans le cache, il n'y a rien d'autre à traiter et le cache n'est pas modifié. Les résultats peuvent être retournés tels quels dans le cas (1), une sous-partie dans le cas (2). Dans le cas (3), le cache contient une ou plusieurs régions contenant la requête Q. Une partie des réponses est donnée par l'union des résultats du cache de cette région, l'autre partie complémentaire doit être traitée hors du cache. Le cache est ensuite réactualisé avec les nouveaux résultats. Dans le cas (4) il y a intersection entre les régions du cache et la requête. on ne récupère donc depuis le cache qu'une partie de l'union des régions et on traite extérieurement la partie complémentaire. Une nouvelle région sera ensuite ajoutée au cache. Enfin, si (cas 5) les régions du cache et la requête sont disjointes la requête doit être posée telle quelle à l'adaptateur et le cache est rempli avec cette nouvelle région.

6.3.4 Politique de mise à jour du cache

La politique de rotation des données conservées dans le cache varie suivant le type de cache. Globalement, nous retiendrons que les types de critères concernant les données à conserver en priorité sont :

1. Le plus fréquemment utilisé.
2. Le plus récemment utilisé.
3. Le plus bénéfique.

Les cas ① et ② sont utilisés généralement dans les caches par page ou par tuple. Le cas ③ est utilisé dans le cache sémantique : on se base sur les notions de « super-requête » (une requête qui englobe le plus de requêtes possible) pour déterminer la requête la plus bénéfique.

6.3.5 Synthèse

Il existe de différentes sortes de caches : des caches de réplication de données, des caches à base d'identifiants de données et des caches sémantiques. Ces derniers sont des caches « intelligents », puisqu'ils exploitent la sémantique des requêtes précédemment formulées afin d'évaluer si une requête doit être entièrement, partiellement ou pas du tout être envoyée sur le réseau. Dans ce domaine encore, si beaucoup de travaux sur les caches dans une architecture de médiation ont été réalisés, peu se sont intéressé aux données semi-structurées.

donnée dans cache mémoire $<$ donnée sur cache disque $\leq$ donnée sur cache source distante $<$ donnée sur disque distant.

6.4 Techniques de stockage de données XML

Pour gérer un cache, il faut mémoriser des documents XML. Ceci pose le problème du stockage de documents XML en mémoire, voire sur des disques au niveau du médiateur. Plusieurs solutions de stockage plus ou moins orientées document ou données existent :

- *stockage sans décomposition en BLOB* : il s'agit du stockage du document sous forme de « données brutes » dans un SGBD classique ou dans un système de fichier, un peu comme les fichiers HTML sur le web. Si le stockage et le chargement permet de retrouver intégralement le document, il est très difficile de faire des requêtes dessus. Ce stockage est orienté document ;
- *stockage sous forme décomposée relationnelle* : il s'agit de faire correspondre certains ou tous les attributs d'un document XML à des tables relationnelle existante. Le stockage est dans ce cas orienté données ;
- *stockage dans une base de données objet* : il s'agit de stocker un document XML comme des objets persistants ;
- *stockage brut dans une base de données native semi-structurée* : dans ce cas, un SGBD est dédié au stockage et aux requêtes sur des données semi-structurées. Il peut s'agir soit d'une extension d'un SGBD existant par le constructeur (IBM DB2, Oracle 9i), soit d'un SGBD entièrement écrit dans cette optique.

Avant de s'interroger sur la manière de stocker des données semi-structurées, il convient de savoir dans quel but on veut les stocker, et quel utilisation on en fera. Ainsi, si le stockage n'a qu'un but simplement documentaire, un simple stockage dans un système de fichier suffirait, par contre, une requête sur les valeurs internes de ce document sera impossible ou très coûteux. De manière inverse, si le document semi-structurée à stocker ne sert de support qu'à certaines informations bien précises sur laquelle on pourra effectuer des requêtes, on peut extraire ces informations au moment du stockage, et les insérer dans une table relationnelle. Dans ce cas, les requêtes sur les informations seront faisables, mais il sera impossible de recréer le document original. Dans le premier cas, le stockage est *centré document* (*document-centric*) et dans le second *centré donnée* (*data-centric*) [Bourret 1999]

Enfin, on peut vouloir réunir tous ces objectifs en privilégiant plus ou moins l'un ou l'autre de ces aspects.

Dans le cadre d'un cache pour des requêtes sur des données semi-structurées nous ne nous intéresserons qu'au stockage *centré donnée*.

6.4.1 Stockage comme un BLOB

Un BLOB (*Binary Large OBject*) est un terme provenant du monde des bases de données et signifiant une séquence d'octets représentant des données. De ce fait, c'est une information non-structurée, qui est manipulée comme un objet opaque pouvant représenter n'importe quelle donnée.

Un stockage de cette façon que ce soit dans une base de donnée en tant que BLOB ou dans un système de fichiers, permet de stocker et charger efficacement un document XML sans perte d'information. Par contre, il sera difficile d'appliquer des requêtes sur la structure et les valeurs du document sans structures annexes (index). Seule des recherches sur mots-clefs coûteuses pourront généralement être effectuées à l'aide d'index spécialisés.

Cette technique de stockage est plutôt adapté à un stockage *centré document*.

6.4.2 Stockage dans une base de données relationnelle

Stocker un document XML dans une base de données relationnelles présente des difficultés ; en effet tout d'abord il s'agit de pouvoir représenter la structure arborescente d'un document XML dans des tables. Il faut aussi gérer les éléments de données et attributs XML sous formes d'attributs relationnels de différents types.

Il y a deux façons de stocker de manière intelligente un document XML dans des tables relationnelles.

La première façon est appelée *mapping générique* [Sha *et al.* 1999]. Elle consiste à utiliser des tables permettant de représenter les nœuds et les liaisons à l'aide de tuples dans des tables internes. Des identifiants uniques (OID) sont associés à chaque élément du documents. Un tableau de liens permet ensuite d'enregistrer les liaisons entre deux éléments, et un tableau de donnée enregistre les valeurs des éléments. Ce type de mapping est transparent à l'utilisateur en permettant d'enregistrer n'importe quel document XML. Par contre, les tables étant codées sous un format interne (tableau de liens et de données), les données ne sont pas exploitables telles quelles sans passer par un mapping inverse.

La seconde façon de stocker des données XML dans un SGBDR est appelée *mapping applicatif*. Elle consiste à mettre en correspondance des parties de documents XML dans des tables relationnelles prédéfinies par l'application. Ce type de mapping nécessite l'utilisation d'outils de mise en correspondance pour chaque nouveau type de document à insérer.

Si ces deux techniques de stockage permettent de gérer facilement des données de structures complexes, la déstructuration (resp. restructuration) du document XML de-

puis (resp. vers) une base de données relationnelle peut s'avérer coûteuse. Mais l'emploi des bases de données relationnelles peut s'avérer nécessaire compte tenu du nombre de systèmes déjà installés ou des applications déjà mises en place, et on peut vouloir bénéficier de la stabilité et des connaissances et optimisations des SGBDR.

6.4.3 Stockage natif

Afin de pouvoir gérer ce nouveau type de données que sont les données semi-structurées, une des solutions serait de créer de toute pièce un nouveau SGBD entièrement dédié à ce type de donné. PDOM [Huck *et al.* 1999], NatiX [Kanne et Moerkotte 2000] ou encore TAMINO [Schning et Wsch 2000] sont des exemples de systèmes natifs. L'avantage d'une telle approche est d'être optimisée pour ce type de données et de fournir une interface d'interrogation efficace et précise. L'inconvénient d'une telle approche est qu'il faut ré-inventer des nouveaux concepts et algorithmes de stockage/récupération pour prendre en charge ces structures nouvelles.

Ainsi, des problèmes nouveaux qui posés. Ces problèmes sont essentiellement causés par :

- *l'absence de schéma* : comment organiser efficacement le placement des données lorsque le schéma n'est pas forcément défini à l'avance ;
- *le volume important des données à stocker* : les documents XML comportent des informations redondantes (des documents de même type ont les mêmes étiquettes qui sont répétées dans chaque document) ;
- *le peu de sélectivité* : due à la diversité des étiquettes qui peuvent être employées en tant que métadonnées, la sélectivité d'un élément peut être très faible.

6.4.4 Synthèse

Il existe plusieurs façons de stocker des données XML : sous forme brute (BLOB), dans un SGBD-R/O ou dans un SGBD natif semi-structuré. Le stockage par BLOB convient à un stockage axé texte, c'est-à-dire de type documentaire. Le stockage SGBD-R convient plutôt à un stockage axé données. Et le stockage dans un SGBD natif peut convenir aux deux types de stockage suivant les spécificités du SGBD natif utilisé.

6.5 Utilisation d'un entrepôt XML comme cache

Si nous établissons une hiérarchie des temps d'accès, nous pouvons constater que du plus rapide au moins rapide est l'accès à la mémoire primaire, les accès disque et

enfin les accès réseaux. Dans le cadre d'un médiateur accédant à des sources distantes sur l'Internet, il est donc intéressant de privilégier les accès locaux (en mémoire ou sur disque). Pour cela, il s'agit de pouvoir représenter l'information des sources distantes sur un système local. Pour cela, une des possibilités est d'utiliser un SGBD local.

Certains systèmes de médiation de bases de données fédérées ont recours à un SGBD local comme cache. Ces SGBD peuvent être relationnel ou objet. Comme nous traitons de données semi-structurées, le plus évident est d'utiliser une base de données gérant des données semi-structurées comme cache. De par sa nature, le cache doit être le plus efficace possible pour stocker et retrouver des objets. Utiliser une base de données semi-structurée non-native, s'appuyant par exemple sur une base de données relationnelle, serait très coûteux en temps d'éclatement/reconstruction. De plus, un cache n'ayant pas à effectuer de requêtes très complexes, il est inutile de s'encombrer de la lourdeur d'un SGBD.

Notre approche est d'utiliser un SGBD natif adapté et minimum. Les seules opérations qui lui sont nécessaires est de savoir stocker puis retrouver une donnée semi-structurée suivant un identifiant unique qu'il affectera.

Ce choix a été fait pour les raisons suivantes :

- les données doivent pouvoir être stockées et extraites sous leur forme première (arborescente) le plus rapidement possible ;
- l'accès à un sous-nœud d'une donnée stockées doit être réalisable rapidement ;
- des opérations sur les chemins peuvent être appliquées et doivent donc être optimisées ;
- pour une bonne optimisation, les chemins, les étiquettes et les valeurs doivent être suffisamment indexés.

Le cache se compose de deux composants : d'une base d'historique permettant de faire la relation entre les requêtes déjà posées et les résultats associés, et un entrepôt de données XML. Nous utilisons comme entrepôt, l'entrepôt natif *ReposiX*.

La figure 6.2 décrit l'intégration du cache dans le médiateur.

La *base d'historique* ① est un tableau comportant :
- la requête ;
- un ensemble d'identifiants des nœuds répondant à cette requête, contenus dans la base locale ;
- la date à laquelle la requête a été enregistrée.

Nous introduirons donc un SGBD natif de données semi-structurées dans notre architecture de médiation. Celui-ci est basé sur le SGBD ReposiX que nous avons introduit dans le chapitre 5.

L'entrepôt natif *ReposiX* (l'architecture a été exposée dans la figure 5.1 du cha-

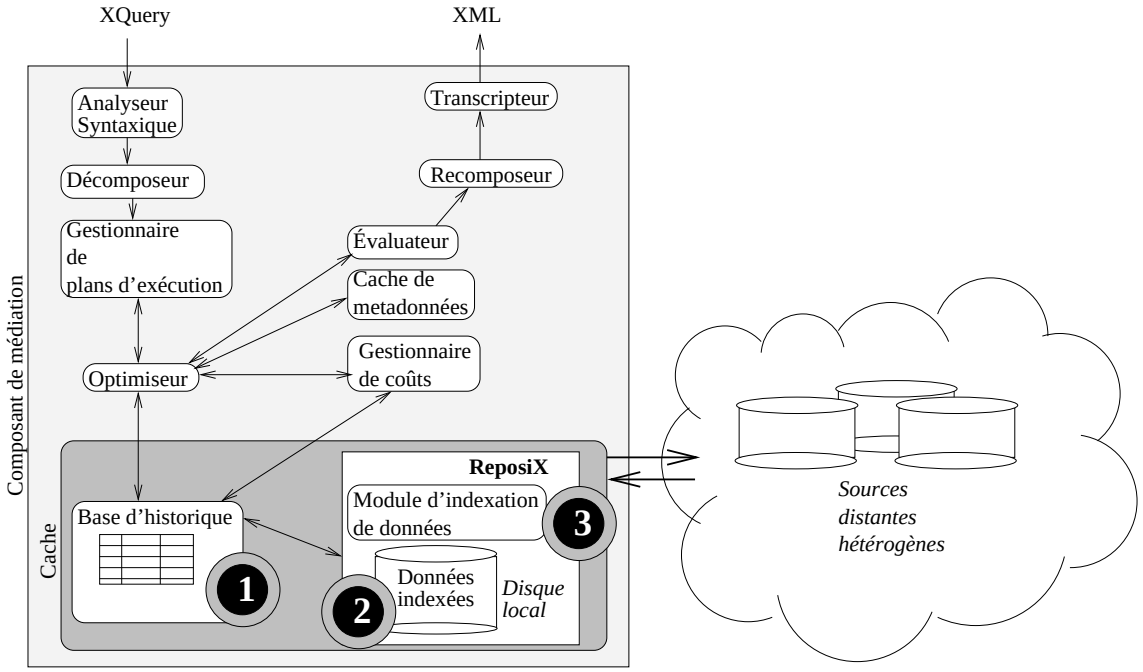


FIG. 6.2 – Intégration du cache dans le médiateur

pitre 5) comporte principalement un composant d'indexation et un composant de stockage/chargement de données semi-structurées. Le composant d'indexation ③ comporte plusieurs index (valeurs, étiquettes, attributs), dont un composant permettant de retrouver rapidement l'ensemble des nœuds associés à un chemin. Le composant de stockage (resp. récupération) de données ②, permet de sérialiser (resp. désérialiser) les arbres DOM qui lui sont passés.

Beaucoup d'opération de stockage et de chargement de données peuvent être réalisées dans ReposiX, pour cela un système de gestion de stockage efficace sur disque a été implémenté.

La figure 6.3 montre l'organisation du système de gestion de stockage de ReposiX.

L'unité de transfert élémentaire entre le disque et la mémoire primaire est appelé un *bloc*. Un *bloc* est alloué sur des régions adjacentes du disque. Un *paquet* est un groupe de blocs contigus. Lors de l'initialisation de ReposiX de larges régions contiguës du disque nommées *fichier* sont allouées par un système spécifique sous-jacent. ReposiX décompose ces régions en nouveau segment. Nous introduisons aussi les *espaces de stockage* consistant en un ou plusieurs *fichiers*. Leurs rôles est de fournir à la demande, des paquets avec un nombre spécifié de blocs contigus. Plusieurs règles ont été introduites afin de réduire l'indisponibilité de paquets de taille spécifique. Chacune de ces règles dicte entre autres, où trouver l'espace pour une liste grandissante (pour la gestion d'index), comment partitionner un objet entre des fichiers, et comment migrer toute ou une partie de cet objet

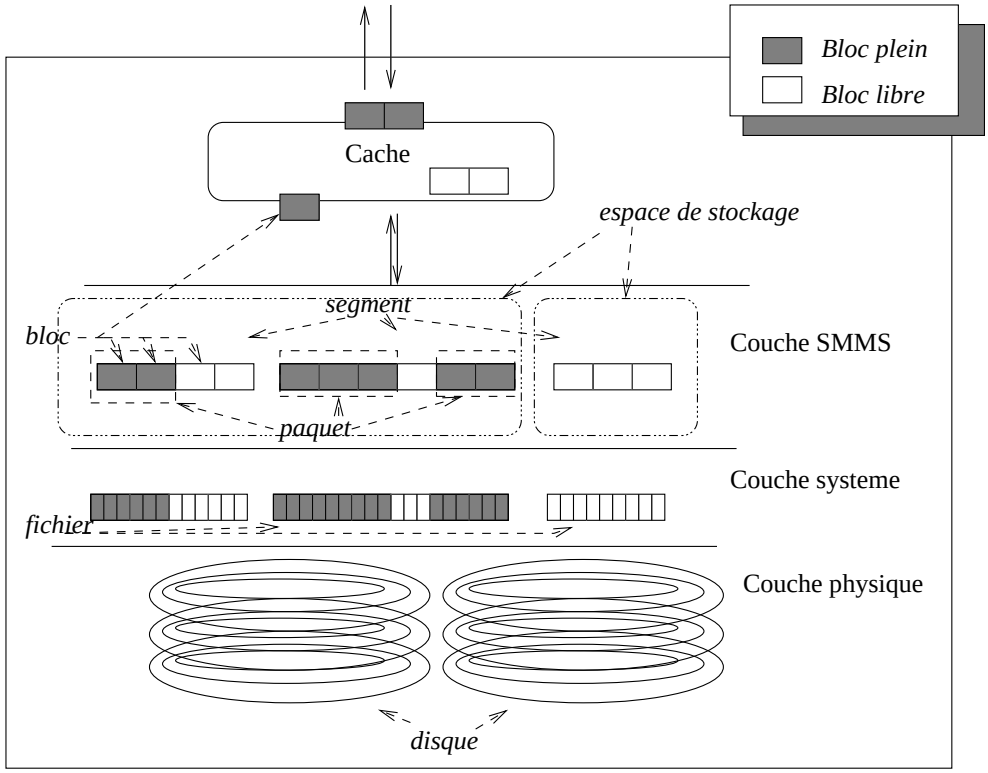


FIG. 6.3 – Organisation du système de gestion de mémoire secondaire de ReposiX

vers l'endroit le plus approprié sur le disque.

Toute cette gestion est réalisée par le système de *gestion de la mémoire secondaire Secondary Memory Management System (SMMS)*, interagissant directement avec le *cache* de ReposiX.

L'indexation des données est faite « à la volée », et est capable de référencer un large volume de données. Les mises à jours étant faites sans reconstruire complètement tout l'index. De façon à optimiser la gestion de l'espace disque, et donc les accès disques, nous considérons aussi les fréquences de mots (rares ou communes) et les mots non-significatifs (articles, particules).

Plusieurs index sont implémentés, un index textuel, d'étiquettes, et aussi un index de chemins et d'identifiants.

ReposiX utilise un identifiant (EID) unique pour chacun des éléments. (exemple voir figure 6.4).

Ainsi sur la figure l'élément de l'instance *personne* est identifié par l'identifiant 318. Cet élément comporte d'autres éléments *nom* (319), *age* (320), *prenom* (321), *adresse*

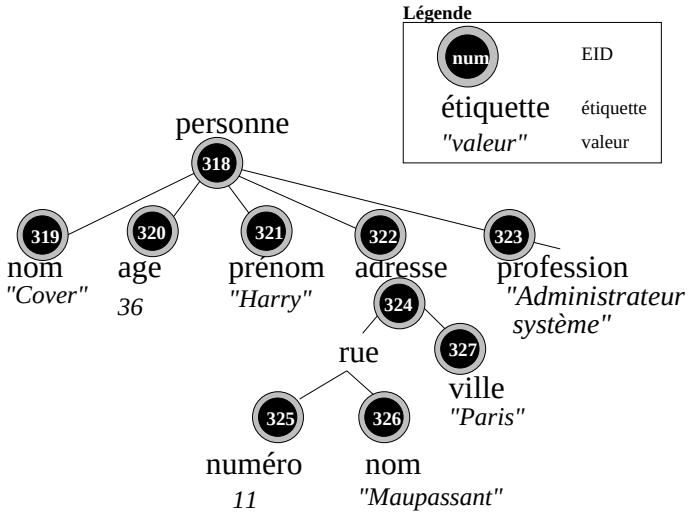


FIG. 6.4 – Identifiants d'éléments

(322) et *profession* (323), etc.

En bref, ReposiX est capable de stocker efficacement des documents XML, et de retrouver rapidement les éléments suivants des critères donnés. Les points qui nous a fait choisir ReposiX comme composant du cache du médiateur sont :

- le stockage compact des documents ;
- ReposiX est capable de renvoyer rapidement un document ou une sous-partie d'un document à partir d'un identifiant donné grace à une indexation des identifiants ;
- la gestion du cache interne de ReposiX, permet de garder en mémoire primaire, un certain nombre de document suivant une politique du « plus fréquemment accédé » ;
- la connaissance du code de ReposiX dont nous avons eu à participer à la conception et au développement lors du projet ESPRIT MUSE.

L'évaluation d'une requête se fait suivant le processus suivant (cf. figure 6.5).

Une requête passée ① à l'évaluateur est analysée, et le la ou les sous-requêtes résultantes est (sont) envoyée(s) ② au cache sémantique. La base d'historique est consultée. Si la requête s'y trouve l'ensemble des identifiants des éléments y répondant est récupéré dans un tableau d'identifiants *tab_eid*. Ce tableau d'identifiants est ensuite passé ③ à ReposiX afin de charger les documents correspondants. Chacun des identifiants est cherché en mémoire primaire. Si il s'y trouve, l'arbre correspondant est renvoyé. Sinon il est chargé ④ depuis la mémoire secondaire (disque local).

Si la requête ne se trouve pas dans la base d'historique, la requête est exécutée ⑤ sur la source distante *via* l'adaptateur correspondant. Le résultat est renvoyé après avoir

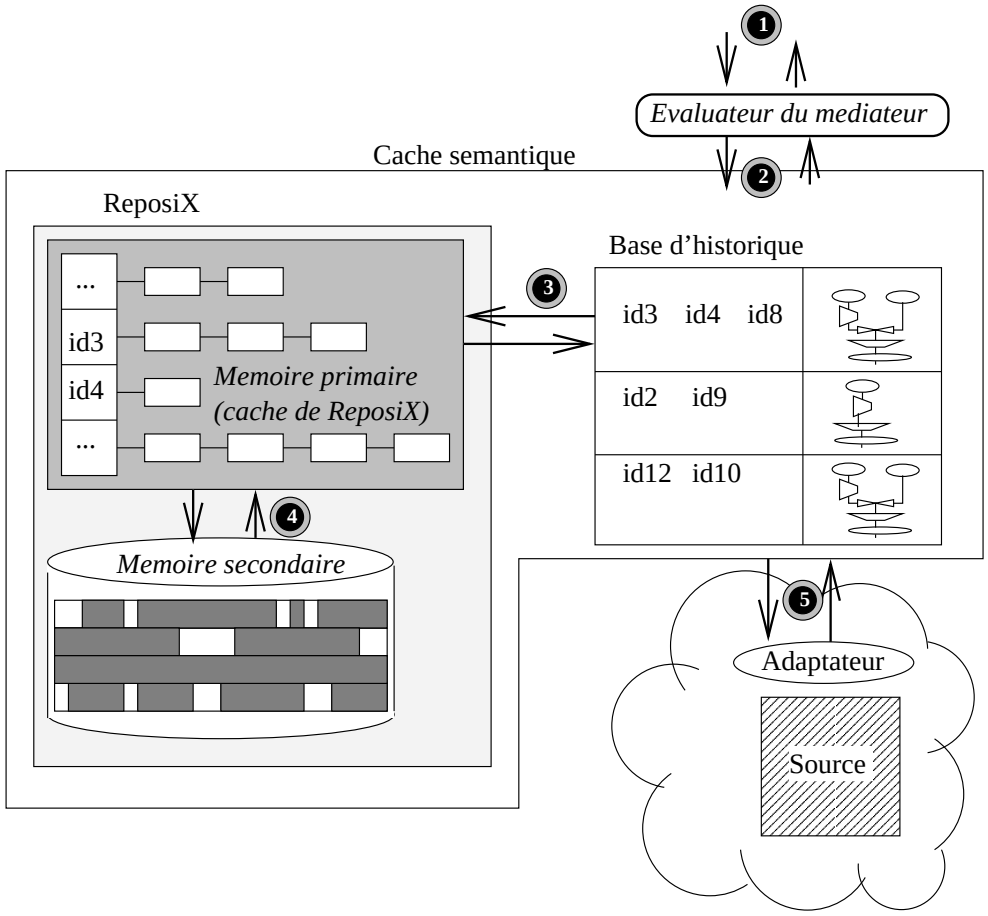


FIG. 6.5 – Évaluation de requête par un cache utilisant ReposiX

été stocké dans le cache sémantique.

L'algorithme précédent s'écrit en pseudo-code objet :

```
methode cache_semantique::executer_requete ( requete) -> xml []
        declarer_tableau  tab_xml : xml []
        si requete est dans le cache semantique  alors
                tab_eid := cache_semantique.executer_requete ( requete)
                tab_xml := reposix.charger ( tab_eid)
        sinon
                tab_xml := adaptateur.executer_requete ( requete)
               cache_semantique.stocker ( tab_xml)
        fin_si

        renvoyer tab_xml
fin_methode

methode reposix:charger ( tab_eid) -> xml []
        tab_xml := initialiser_tableau_xml ()
        pour chaque  i  de tab_eid
                si  tab_eid [i] est dans cache_primaire  alors
                        tab_xml [i] := cache_primaire.charger ( tab_eid [i])
                sinon si  tab_eid [i] est dans cache_secondaire  alors
                        tab_xml [i] := cache_secondaire.charger ( tab_eid [i])
                sinon
                       erreur
                fin si
        fin_pour
        renvoyer  tab_xml
fin_methode
```

6.6 Gestion d'un cache sémantique

Nous décrivons ici la gestion d'un cache sémantique, c'est-à-dire un cache prenant en compte la signification de la requête. Nous verrons comment classifier les requêtes et leurs critères de recouvrement. Nous verrons comment est organisée la base d'historique et sa politique de mise à jour ainsi que son interaction avec l'entrepôt ReposiX.

6.6.1 Requête, sous-requête et super-requête

Intuitivement, une requête comme :

```
-- Chercher tous les livres dont le titre contient le mot « LINUX ».
SELECT *
FROM livre
WHERE
       titre LIKE '%LINUX%'
```

est beaucoup moins restrictive que :

```
-- Chercher tous les livres dont le titre contient le mot « LINUX » et
-- qui soit paru après 1989.
SELECT *
FROM livre
WHERE
       titre LIKE '%LINUX%'  AND date > 1989
```

qui est elle même moins restrictive que :

```
-- Chercher tous les livres dont le titre contient le mot « LINUX »,
-- qui soit paru après 1992
SELECT *
FROM livre
WHERE
       titre LIKE '%LINUX%'  AND date > 1992
```

Définition 6.1 : Restrictivité

Une requête R est dite plus *restrictive* qu'une requête R', si et seulement si l'ensemble des résultats de R est inclus dans l'ensemble des résultats de R'.

6.6.2 Modèle et notation d'un cache à base de prédicats

Nous allons à présent formaliser notre terminologie.

Soit un médiateur supportant n adaptateurs. Soit A_i le i^{eme} adaptateur, $1 \leq i \leq n$. Soit Q_i le nombre de requêtes dans le cache correspondant à l'adaptateur A_i, $Q_i \geq 0$. On écrit $P_{i,j}$ le j^{eme} prédicat de requête (where) caché correspondant à l'adaptateur A_i, $0 \leq j \leq Q_i$. On appellera $\mathcal{R}(P_{i,j})$ l'ensemble des résultats du prédicat $P_{i,j}$.

Définition 6.2 : Cache

Le cache pour le i^{eme} adaptateur A_i est défini comme l'ensemble des prédicats de requêtes $P_{i,j}$ correspondant à tous les résultats de requêtes cachées.

$$C_i = \{P_{i,j} | 1 \leq i \leq n, 1 \leq j \leq Q_i\}$$

Définition 6.3 : Contenance de requête

On dira qu'un prédicat de requête P_x est *contenu* dans un prédicat de requête P_y et on notera $P_x \sqsubseteq P_y$ si et seulement si pour toute base de données B l'ensemble des résultats $\mathcal{R}(P_{B,x})$ répondant au prédicat P_x est *inclus* dans l'ensemble des résultats $\mathcal{R}(P_{B,y})$ répondant au prédicat P_y.

$$P_x \sqsubseteq P_y \; ssi \; \mathcal{R}(P_x, P) \subseteq \mathcal{R}(P_y, P)$$

Définition 6.4 : Requêtes équivalentes

Deux requêtes P_x et P_y sont dites *équivalentes* et noté $P_x \equiv P_y$ si $P_x \sqsubseteq P_y$ et $P_y \sqsubseteq P_x$.

6.6.3 Restrictions sur les relations

À chaque relation, on associe un ensemble de modèle de limitation (*binding pattern*). Formellement, un modèle de limitation pour une relation R est une correspondance entre les arguments de R et l'alphabet $\{b, f\}$. Un attribut auquel correspond b précise que la relation doit fournir une constante pour cet attribut. Par exemple le modèle de limitation $R(a^b, b^f)$ pour la relation $R(a, b)$ spécifie un modèle où les valeurs de a doivent être données pour obtenir les tuples de R. Ces notations sont bien connues dans le contexte de la programmation logique (DATALOG). Nous les utiliserons ici pour résoudre les requêtes sur le cache.

6.6.4 Politique de mise à jour du cache

Soit les notions définies ci-dessous :

Définition 6.5 : Sous-prédicat local, Sous-prédicat complémentaire

Soit P_e un prédicat de requête à exécuter. Soit C_i le cache pour le i^{eme} adaptateur A_i du médiateur. Alors :
- P_e est *exécutable localement* ssi

$$\forall j \in [1, Q_i], P_e \sqsubseteq P_{i,j}$$

- P_e est *partiellement exécutable localement* ssi

$$\exists j \in [1, Q_i] \text{ et } \exists P_e^l \sqsubseteq P_e \ / \ P_e^l \sqsubseteq P_{i,j}$$

P_e^l est appelé le *sous-prédicat local* du prédicat P_e. On appellera *sous-prédicat complémentaire* P_e^c de P_e^l à P_e un prédicat tel que

$$\mathcal{R}(P_e^c) \cup \mathcal{R}(P_e^l) = \mathcal{R}(P_e)$$

- P_e est *aucunement exécutable localement* ssi

$$\nexists j \in [1, Q_i], P_e \sqsubseteq P_{i,j}$$

Si P_e est *exécutable localement*, alors on exécute la requête localement sur le cache.

Si P_e est *partiellement exécutable localement*, alors on exécute le *sous-prédicat local* P_e^l sur le cache, et on exécute le *sous-prédicat complémentaire* P_e^c sur l'adaptateur. P_e^l est remplacé par P_e dans le cache et $\mathcal{R}(P_e^l)$ est complété par $\mathcal{R}(P_e)$

Si P_e est *aucunement exécutable localement*, alors on exécute Si P_e sur l'adaptateur et on remplit le cache avec le prédicat P_e et l'ensemble de résultat $\mathcal{R}(P_e)$.

6.6.5 Organisation du cache

Lorsqu'un arbre est stocké dans l'entrepôt, chacun de ses nœud obtient un identifiant unique le représentant dans l'entrepôt. Le mécanisme d'indexation (basé sur un B+Tree et des listes inversées [Cutting et Pedersen 1990]) et le stockage interne des documents permettent de retrouver rapidement l'ensemble des identifiants des nœuds correspondant à un chemin donné. Connaissant l'identifiant d'un nœud, le mécanisme de stockage de ReposiX permet de récupérer le sous-arbre associé à ce nœud en un minimum d'entrées/sorties.

La base d'historique permet d'associer un domaine de requête à un ensemble d'identifiants répondant à cette requête. Il suffit donc de déterminer lorsqu'une nouvelle requête est formulée, dans quelle mesure une requête déjà stockée peut répondre à cette dernière.

Ce mécanisme peut se résumer dans le tableau suivant. Soit Q la requête utilisateur et R une vue sémantique du cache.

Relation entre Q et R	Propriétés	Résultat du cache	Résultat des adaptateurs
Équivalence	$Q \equiv R$	R	$\emptyset$
Contenant	$(Q \sqsubseteq R) \wedge (R \not\sqsubseteq Q)$	$Q(R)$	$\emptyset$
Contenu	$(Q \not\sqsubseteq R) \wedge (R \sqsubseteq Q)$	R	$Q \wedge \neg R$
Intersection	$(Q \not\sqsubseteq R) \wedge (R \not\sqsubseteq Q) \wedge (R \bigcap Q = \emptyset)$	$Q(R)$	$Q \wedge \neg R$
Disjonction	$Q \bigcap R = \emptyset$	$\emptyset$	Q

On distingue les cas où le cache peut répondre entièrement à la requête (requête équivalente, ou requête contenue dans une requête du cache) ; le cas où le cache peut répondre partiellement à la requête (requête contenant une requête du cache ou ayant une intersection commune), dans ce cas une requête complémentaire aux adaptateurs est nécessaire ; et enfin le cas où le cache ne répond pas du tout à la requête (requête disjointe), et dans ce cas elle peut être envoyée telle quelle aux adaptateurs.

Il suffit donc de définir comment déterminer les relations entre le domaine d'une requête Q et le domaine d'une autre requête R. Pour cela, des règles de restrictivité des requêtes doivent être formulées, c'est ce que nous verrons dans la section suivante.

6.6.6 Règle de détermination de restrictivité

Nous avons défini précédemment qu'un prédicat P_1 est *contenu* dans un prédicat de requête P_2 ($P_1 \sqsubseteq P_2$) si et seulement si pour toute base de données B l'ensemble des résultats $\mathcal{R}(P_1, B)$ répondant au prédicat P_1 est *inclus* dans l'ensemble des résultats $\mathcal{R}(P_2, B)$ répondant au prédicat P_2. Or, au moment de l'étude de la requête, on n'a pas encore évalué l'ensemble des résultats. Il faut donc déterminer la restrictivité à partir de la sémantique même des deux prédicats P_1 et P_2.

La contenance de requête dans le cas des requêtes conjonctives (sélection, projection, jointure) est un problème NP-complet [Chandra et Merlin 1977]. Mais on peut déterminer des algorithmes à temps bornés pour certains cas spéciaux [Ullman 1989]. Les travaux faits à ce sujet, notamment [Luo *et al.* 2001], [Adali *et al.* 1996] et [Chidlovskii *et al.* 1999] ont énuméré des règles de restrictivité des requêtes, principalement :

Les prédicats utilisant les opérateurs simples de restriction ($\geq, \leq, \geq, \leq, =, \neq, LIKE$) sont des prédicats simples. Et une comparaison de deux requêtes comportant de tels

prédicats est possible d'après leurs domaines de définition des résultats.

Dans le cas de AND ou de OR, [Luo *et al.* 2001] propose certaines règles de détermination de restrictivité :

Proposition 1 Soit les conditions WHERE telles que :

$$conditionWhere1 = P_1\ AND\ P_2\ AND...P_m$$

$$conditionWhere2 = Q_1\ AND\ Q_2\ AND...Q_n$$

alors la conditionWhere1 est plus restrictive que la conditionWhere2 ssi :

$$\forall i, 1 \leq i \leq n, \exists k, 1 \leq k \leq m, P_k\ est\ plus\ restrictif\ que\ Q_i.$$

Proposition 2 Soit les conditions WHERE telles que :

$$ConditionWhere1 = P_1\ OR\ P_2\ AND...P_m$$

$$ConditionWhere2 = Q_1\ OR\ Q_2\ OR...Q_n$$

alors $ConditionWhere1$ est plus restrictive que $ConditionWhere2$ ssi :

$$\forall i, 1 \leq i \leq m, \exists k, 1 \leq k \leq n, P_i\ est\ plus\ restrictif\ que\ Q_k.$$

Nous proposons d'étendre ces règles à des prédicats portant sur les chemins. L'entrepôt natif de données semi-structurées que nous utilisons utilise un identifiant (EID) unique pour chacun des éléments.

Si l'on pose une requête sur un sous-chemin de chemins déjà stockés dans le cache, moyennant certaines conditions, il n'est pas besoin de formuler à nouveau la requête à l'adaptateur.

Soit une requête R de type :

for $\$x$ **in** $\mathcal{C}_R$
where $\mathcal{P}_R(\$x)$
return $\mathcal{C}'_R(\$x)$

Où $\mathcal{C}_R$ et $\mathcal{C}'_R$ sont des chemins, et $\mathcal{P}_R$ un prédicat. Un chemin $\mathcal{C}'_R$ est un ensemble d'étiquettes connexes du graphe, et s'écrit : $\mathcal{C}'_R = (c'_{R,1}, c'_{R,2}, ..., c'_{R,n})$ où n est la *longueur* du chemin $\mathcal{C}'_R$.

On regarde dans le cache une requête R_{ci} déjà résolue similaire à cette dernière. Nous allons décomposer la requête R en deux parties à étudier :

1. La structure résultat (clause « return »).
2. La structure prédicat (clause « where »).

Le premier point est le premier critère à étudier. Il faut rechercher dans le cache s'il n'existe pas de requête R_{ci} dont le chemin $\mathcal{C}_{ci}$ aurait un préfixe commun avec $\mathcal{C}'_R$. Soit $p = \mathcal{C}'_R \dot{\sqcap} \mathcal{C}_{ci}$ le plus grand préfixe commun entre $\mathcal{C}_{ci}$ et $\mathcal{C}'_R$. Soit l la longueur du préfixe p. On a donc :

$$p = (c'_{R,1}, c'_{R,2}, ..., c'_{R,l}) = (c_{c,1}, c_{c,2}, ..., c_{c,l})$$

puisque dans $\mathcal{C}_{ci}$ comme dans $\mathcal{C}'_R$, les l premières étiquettes sont identiques et correspondent au préfixe.

On distingue alors plusieurs cas :

1. Si le préfixe commun est vide (*ie.* $p = \emptyset$) : $\mathcal{C}'_R$ et $\mathcal{C}_{ci}$ n'ont pas de préfixe commun, $\mathcal{C}'_R$ n'est donc pas dans le cache et la requête entière doit être envoyée vers l'adaptateur.
2. Si le préfixe commun est égal à $\mathcal{C}'_R$ **et** à $\mathcal{C}_{ci}$ (*ie.* $p = \mathcal{C}'_R = \mathcal{C}_{ci}$) : $\mathcal{C}'_R$ et $\mathcal{C}_{ci}$ sont *structurellement équivalents*. Il faut donc regarder au niveau des prédicats afin de déterminer quelle requête doit être envoyée aux adaptateurs et quelle partie peut être traitée dans le cache.
3. Si le préfixe commun est égal à $\mathcal{C}'_R$ (*ie.* $p = \mathcal{C}'_R$), $\mathcal{C}'_R$ est structurellement contenu dans $\mathcal{C}_{ci}$ du cache. Il faut donc regarder au niveau des prédicats afin de déterminer quelle requête doit être envoyée aux adaptateurs et quelle partie peut être traitée dans le cache.
4. Si le préfixe commun est égal à $\mathcal{C}_{ci}$ (*ie.* $p = \mathcal{C}_{ci}$), $\mathcal{C}'_R$ contient structurellement $\mathcal{C}_{ci}$ du cache. Dans ce cas, la requête à adresser aux adaptateurs est :

   for $\$x$ **in** $\mathcal{C}_R$
   where $\mathcal{P}_R(\$x)$
   and not exists $(C'_R(\$x))/c_{ci,(l+1)}$
   return $\mathcal{C}'_R(\$x)$

   La requête complémentaire R_{comp} est structurellement résoluble par le cache, mais il faut regarder au niveau des prédicats afin de déterminer quelle requête doit être envoyée aux adaptateurs et quelle partie peut être traitée dans le cache.

Lorsque une requête est structurellement résoluble par le cache (R pour les cas 2 et 3 et R_{comp} pour le cas 4), il faut s'appuyer sur le prédicat pour déterminer dans quelle mesure la requête peut être traitée par le cache. Sur cette partie-là, nous rejoignons le cas classique de la résolution de la restrictivité dans les caches sémantiques traditionnels.

Ceci se résume par le pseudo-algorithme suivant :

```
 fonction evaluation (Requete requete, Cache cache)
XPath prefixe := calculer_plus_grand_prefixe_commun ( prédicat1, requete)
XPath chemin1 := récupérer_chemin (requete)
XPath chemin_c := récupérer_chemin (cache, prefixe)
Requete requete_locale
Requete requete_adaptateur
 // 1) chemin1 et chemin_c disjoints
 si est_vide (prefixe)
     requete_locale := null
     requete_adaptateur := requete

 // 2) chemin1 et chemin_c égaux
 sinon si prefixe = chemin1 = chemin_c
     requete_locale := calculer_predicat_local requete)
     requete_adaptateur := calculer_predicat_adaptateur requete)

 // 3) chemin_c inclut chemin1
 sinon si prefixe = chemin1 != chemin_c
     requete_locale := calculer_predicat_local requete)
     requete_adaptateur := calculer_predicat_adaptateur requete)

 // 4) chemin1 inclut chemin_c
 sinon si prefixe != chemin1 = chemin_c
     requete_locale := calculer_predicat_local requete)
     requete_adaptateur := calculer_predicat_adaptateur requete)
    reecrire_projection (requete)
 fin si
retourner ( requete_locale, requete_adaptateur)
 fin fonction
```

6.7 Extension au modèle de coût

Le fait d'utiliser un cache sert à réduire les exécutions multiples de la même requête. Un cache permet de réduire le temps d'exécution d'une requête de deux façons différentes : pendant la phase d'optimisation, les sous-requêtes déjà présentes dans le cache ne sont pas optimisées, permettant ainsi de réduire le temps d'optimisation. Enfin pendant la phase d'exécution, les sous-requêtes qui sont dans le cache ou en partie dans le cache ne sont pas exécutées (ou exécutées partiellement). Ceci réduit le temps d'exécution à la fois en jouant sur le coût des adaptateurs, et sur le coût de communication. En jouant sur un cache secondaire d'exécution, cela réduirait aussi le coût au niveau du médiateur (ex. jointures déjà exécutées).

Ainsi l'utilisation du cache primaire et secondaire changent le coût de la façon suivante :

pour une requête Q de plan d'exécution $\mathcal{P}_Q$.

```
fonction coût (Noeud noeud)
    si est_feuille ( noeud)
        si dans_cache ( noeud)
            retourner coût_cache ( noeud)
        sinon // noeud sur adaptateur
            retourner coût_adaptateur ( noeud))
     fin_si

    si (requête associée au sous-arbre dont la racine est noeud
       est dans le cache primaire ou le cache secondaire)
    alors
        retourner coût_cache ( noeud)
    sinon
        pour chaque fils de noeud
           coût (fils)
        fin_pour
       évaluer (coût du noeud à l'aide de la formule de coût associée
                à ce noeud et le coût de chacun des fils)
        retourner (coût calculé)
    fin_si
fin_fonction
```

Le coût d'un nœud de l'arbre algébrique est égal au coût du nœud plus celui de chacun de ses fils (algorithme récursif). Si un nœud se trouve dans le cache, alors le coût associé est celui associé au coût d'une requête locale dans le cache. Sinon, le nœud terminal est alors évalué par l'adaptateur, et le coût est calculé à l'aide du modèle de coût de l'adaptateur.

6.8 Conclusion

Dans ce chapitre nous avons présenté la manière dont s'intègrerait un cache dans une architecture de médiation dont le modèle interne est semi-structuré. Nous avons montré que le cache devait s'appuyer sur un système pouvant gérer directement les données semi-structurées afin d'être plus performant et éviter des temps de déstructuration/reconstruction inutiles de telles données. Nous nous sommes pour cela appuyé sur l'entrepôt natif de données semi-structurées ReposiX.

Ensuite, nous avons montré comment utiliser un tel entrepôt avec une structure d'historique afin de définir un cache de requêtes (cache sémantique). De nombreux travaux expliquent comment utiliser un cache sémantique sur des données relationnelles, et les relations d'ensemble entre les domaines de résultats de deux requêtes. À notre connaissance, aucun travail à ce jour n'a été fait quand aux relations de domaines de résultats sur des requêtes portant sur des données semi-structurées. Nous avons étudié comment les domaines de requêtes pouvaient être étudiés par rapport à deux requêtes comportant des opérations de navigation. En effet, l'étude de la restrictivité des requêtes en semi-structuré en vue de l'élaboration d'un cache sémantique est complexe puisqu'elle ne fait plus intervenir seulement l'inclusion des prédicats. Il faut en plus tenir compte de l'inclusion des

chemins utilisés et retournés. Ceci ajoute une difficulté supplémentaire dans le domaine de caches

Et enfin nous avons montré comment intégrer le cache dans le modèle de coût.

Chapitre 7

Prototypes

7.1 Introduction

Le médiateur présenté dans cette thèse résulte de l'évolution d'un prototype au travers de trois projets successifs. Au cours de ces trois projets, le médiateur a subit des modifications au fil de l'évolution des technologies et des standards XML.

Les travaux développés dans cette thèse s'appuient sur l'architecture du projet Esprit MIROWEB[1] qui est à l'origine de la création de la société e-XMLMedia, ainsi que sur l'architecture du médiateur *XMLMEDIA* développé au sein de cette société. Ce dernier a servi de composant fédérateur dans le projet Esprit XML-KM[2] (*XML Knowledge Management*) - suite directe du projet MIROWeb- et le projet RNTL MUSE[3] (*Multimedia Search Engine*) - dont l'objectif est de stocker et d'interroger des documents multimédia répartis.

L'objectif du projet MIROWEB est de construire un système d'accès à des sources de données hétérogènes sur le Web. Nous avons pour cela conçu une architecture de médiation à base de médiateur et adaptateurs sur le modèle DARPA I3. Le modèle de données commun est un modèle de type OEM, le langage d'interrogation est XML-QL et les résultats sont exprimés en XML. L'expérimentation a été effectuée sur deux applications pilote : un système d'information hospitalier et une application basée sur le tourisme. Les sources intégrées ont été un entrepôt XML basées sur Oracle 8, et le SGBD relationnel

[1] Les partenaires du projet sont : Ibermática (Espagne), Bull (France), OSIS (France), BHS (Espagne), TIS (Autriche) et l'INRIA (France)

[2] Les partenaires du projet sont : Ibermática S.A (Espagne), e-XMLMedia (France), SITESA (Espagne), TIS GmbH (Autriche), Cámara Oficial de Comercio, Industria y Navegación de Gipuzcoa (Espagne)

[3] Les partenaires du projet sont : Editing, PRiSM, ECL, ICCT, e-XMLMedia - Ces travaux ont été financés en partie par le MENRT dans le cadre du projet MUSE

Oracle 8i. Cette version a constituée la version v0 du médiateur.

Le projet XML-KM consiste en l'intégration d'applications existantes sur le commerce et le tourisme avec un système d'information géographique et un entrepôt de données. Nous sommes pour cela parti des bases du médiateur v0 de MIROWEB, et nous avons fait évoluer la définition des adaptateurs afin de permettre la définition générique d'autres adaptateurs (un adaptateur de descripteur de cartes réalisés par SITESA (Espagne). Cette version a constituée la version v1 du médiateur.

Le projet MUSE a pour objectif la conception et la réalisation d'un moteur de recherche ciblé permettant de découvrir rapidement des données XML pouvant inclure des documents multimédia. La version v1 du médiateur a été réécrite afin de supporter le langage XQuery au lieu de XML-QL, avoir pour modèle commun et modèle résultat XML, coller autant que possible aux normes du W3C. Et définir une interface normalisée d'interaction d'importation et d'exportation entre les adaptateurs et le médiateur. Cette version constitue la version v2 du médiateur.

7.2 Plan du chapitre

Nous présentons dans ce chapitre les trois projets qui ont fait l'objet de cette thèse. Dans la section 7.3, nous présentons le projet MIROWEB. La section 7.4 expose le projet XML-KM. Et enfin la section 7.5 présente le projet MUSE.

7.3 Expérience du projet MIROWEB

Le projet MIROWEB a permis le développement de la version 0 du médiateur au PRiSM.

L'objectif de MIROWEB (projet ESPRIT-25208) est de construire un système d'accès à des sources de données hétérogènes sur le Web, de développer les outils associés, et d'expérimenter sur deux applications pilotes : un système d'information hospitalier et une application basée sur le tourisme.

L'architecture de MIROWEB (figure 7.1) reprend le modèle d'architecture trois-tiers. Au niveau client, nous avons une interface de navigation basée sur XML et une API JDBC. Au niveau intermédiaire, il y a un intégrateur automatique de schémas et deux médiateurs : l'un s'appuyant sur le prototype DISCO de l'INRIA, l'autre s'appuyant sur le SGBD Oracle8 étendue avec une cartouche de gestion de données semi-structurées. C'est ce dernier médiateur que nous avons développé et que nous décrirons ici (figure 7.2).

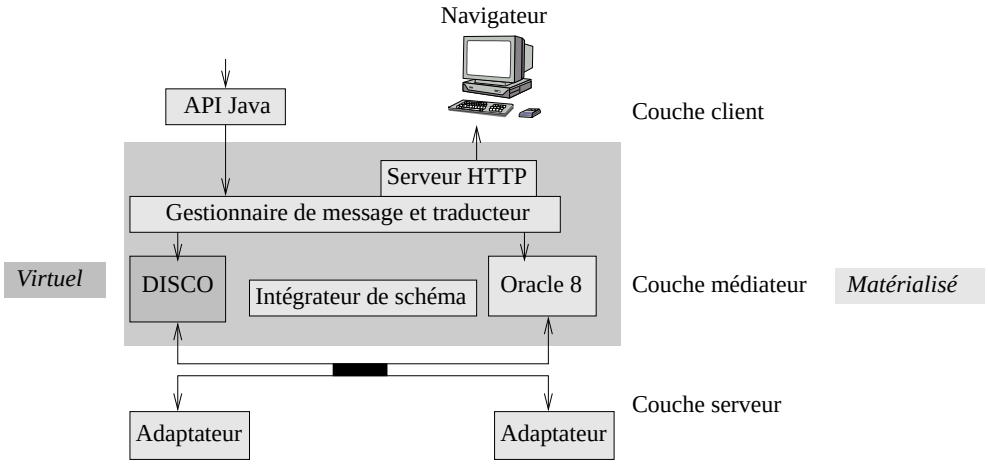


FIG. 7.1 – Architecture du projet MIROWEB

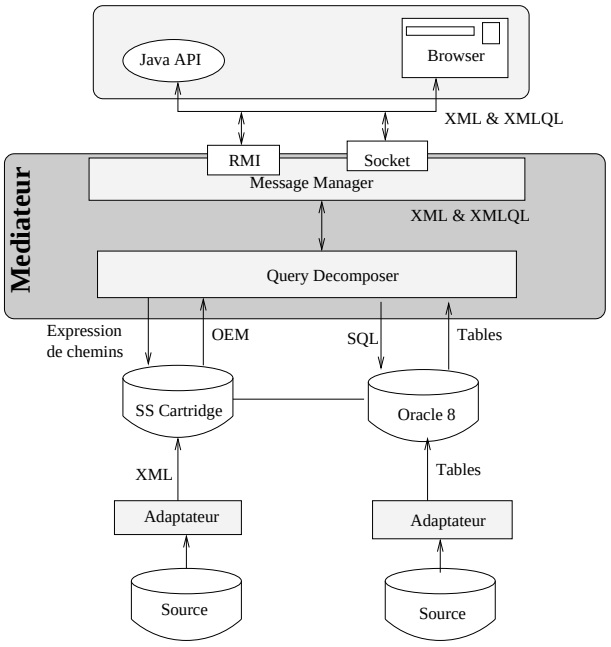


FIG. 7.2 – Architecture du médiateur de MIROWEB

Le médiateur [Gardarin *et al.* 1999] s'appuie sur le langage XML-QL restreint à quelques fonctionnalités. MIROWEB fédère des sources de données hétérogènes en s'appuyant sur un modèle de données semi-structurées. Le modèle de données est une variante du modèle OEM [McHugh *et al.* 1997]. Les données sont échangées entre les sources et le médiateur par l'intermédiaire de XML et XML-QL.

Les sources utilisées sont une cartouche OEM s'appuyant sur un SGBD oracle 8.0

utilisé comme entrepôt de données semi-structurées, ainsi qu'une base de données relationnelles Oracle 8.0. L'adaptateur de données semi-structurées communique avec la cartouche OEM par des appels de primitives basé sur des expressions de chemin et récupère un arbre de type OEM. L'adaptateur de données relationnelles communique avec le SGBD-R en SQL et récupère des ensembles de tuples *via* JDBC. Les composants incluent également un adaptateur XML nommé JEDI utilisant un extracteur de données à base de règles développé par le GMD IPSI. Une implémentation de dataguide a été faite afin de gérer le schéma des différentes sources et permettre à l'utilisateur finale de connaître la structure des données qu'il peut pouvoir interroger. L'évaluation se fait en mettant les documents XML à plat. Cette méthode permet ainsi de pouvoir évaluer les requêtes de la même façon qu'en algèbre relationnelle. L'inconvénient est que dans le cas d'attributs multivalués, cela augmente rapidement le nombre de tuple à traiter et ne conserve pas forcément un groupement cohérent.

7.4 Expérience du projet XML-KM

Le projet XML-KM a permis le développement de la version 1 du médiateur XML-MEDIA dans la société e-XMLMedia.

L'objectif de XML-KM (projet ESPRIT IST-12030) est l'intégration d'applications existantes sur le commerce et de tourisme avec un SIG (Système d'Information Géographique) et un entrepôt de données ainsi que les outils qui ont été développés dans le projet MIROWEB. L'intégration des données du SIG avec les données de l'entrepôt de données est réalisé *via* SVG (*Scalable Vector Graphic*), le futur standard pour le graphisme à deux dimensions en XML. Tout ceci vise à produire une interface uniforme de recherche à travers diverses sources de données basées sur XML.

L'architecture du projet XML-KM (figure 7.3) est composée d'applications clientes, d'un serveur d'application et de serveurs de données.

Les *applications clientes* se composent d'un navigateur, des interfaces de programmation JAVA vers le serveur et un gestionnaire de profil d'utilisateurs.

Le *serveur d'applications* inclut un serveur web, le médiateur d'e-XMLMedia et un moteur de présentation à base de feuilles de style XSL. Toutes les applications utilisent le serveur Web d'Apache et les logiciels libres (*open source*) xalan, xerces. Le médiateur XML fournit les fonctionnalités nécessaires pour traiter les requêtes XML distribuées. Le moteur de présentation XSL est dérivé du logiciel libre d'Apache pour les moteurs XSL appelé Xalan. Si le navigateur ne supporte pas les SVG, XSL/FO peut être utilisé pour les convertir en un format portable standard JPEG et GIF.

Le *serveur de données* inclus des sources de données diverses et des moteurs de

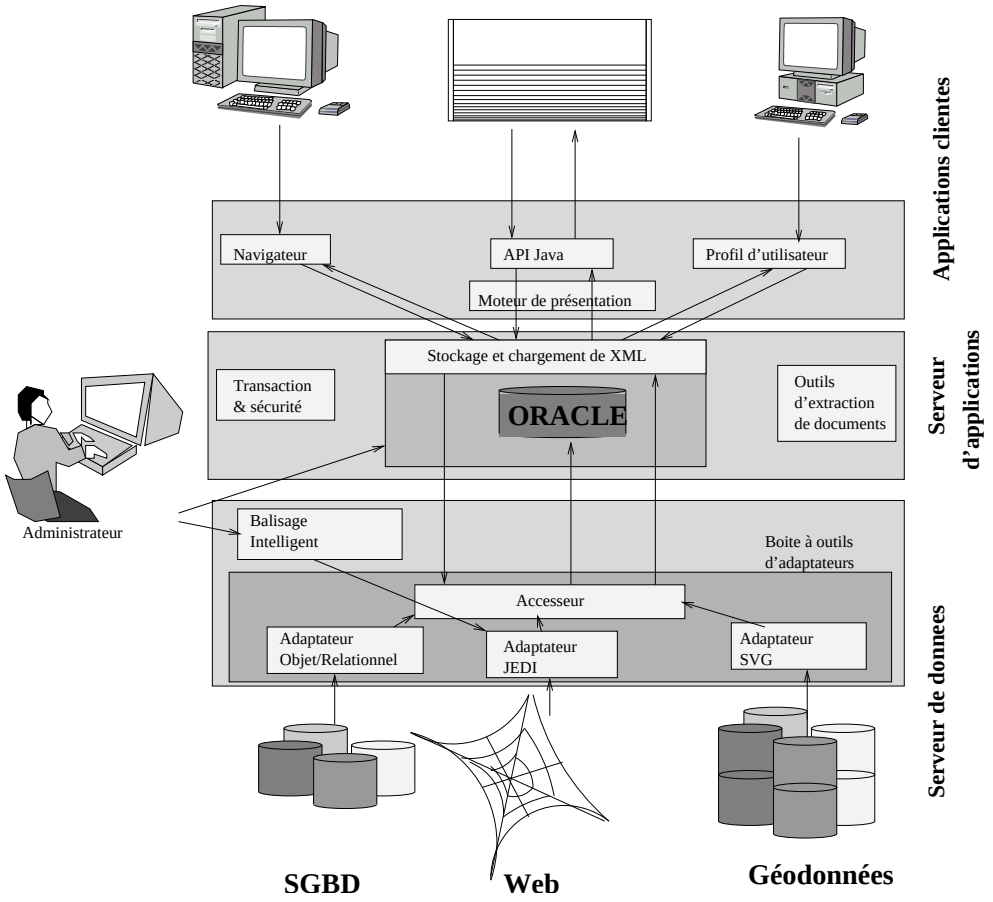


FIG. 7.3 – Architecture du projet XML-KM

traduction. Les données XML sont stockées dans l'entrepôt *repository v1* de e-XMLMedia. L'adaptateur communique avec l'entrepôt par un langage spécifique SQLX et récupère un arbre de type OEM. Pour stocker des données géographiques, le projet utilise l'outil ArcSDE de ESRI qui les stocke dans Oracle. Enfin pour accéder aux SGBD-OR, un adaptateur objet-relationnel est implémenté. L'adaptateur JEDI permet de transformer et chercher des données depuis diverses autres sources.

L'architecture du médiateur est décrite à la figure 7.4. Le médiateur est composé de deux types de processus *multithreads* : l'évaluateur de requête et les passerelles. L'évaluateur reçoit des requêtes XML-QL et les décomposent sous format interne. Il isole ensuite les sous-requêtes locales et les distribuent aux passerelles. Les passerelles s'occupent de l'accès aux bases et des connexions aux adaptateurs. Les requêtes sont traduites dans le format commun pour les adaptateurs et envoyées aux adaptateurs correspondants. Les résultats sont renvoyés par les adaptateurs sous forme de documents XML ou de graphe DOM. Ils sont ensuite assemblés par le moteur de requête qui complète la requête (par exemple les jointures entre les sous-requêtes). Le résultat final est envoyé à l'application cliente sous la forme d'un document XML.

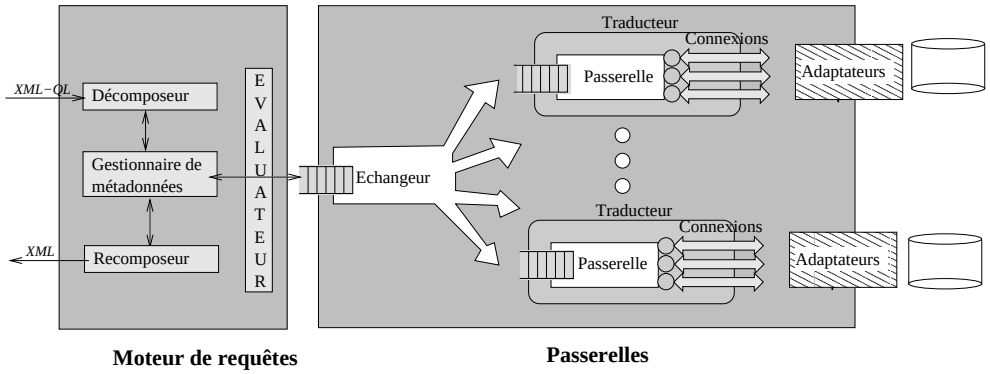


FIG. 7.4 – Architecture du médiateur de XML-KM

7.5 Expérience du projet MUSE

Le projet MUSE a permis le développement de la version 2 du médiateur XMLMEDIA à e-XMLMedia.

Le projet MUSE (projet RNTL) a pour objectif la conception et la réalisation d'un moteur de recherche ciblé permettant de découvrir rapidement des données XML, pouvant inclure des documents multimédia. Le moteur de MUSE est utilisé pour répondre aux besoins de l'agence photographique Editing.

Le moteur de recherche est implémenté comme suit (figure 7.5). L'utilisateur interagit avec une interface Web pour formuler simplement ses requêtes à l'aide du langage XQuery étendu avec des fonctions de recherche plein texte et de similarité. Au cœur du système, un médiateur reçoit les requêtes et assure leur décomposition en requêtes mono-source, la soumission aux sources *via* des adaptateurs, l'intégration et le filtrage de résultats. Un entrepôt natif nommé ReposiX assure le stockage et la recherche de documents par des méthode d'accès spécialisées rapides. Il gère en particulier les descripteurs de contenus (textes, images) en XML s'approchant du standard MPEG7.

Le médiateur est le médiateur actuel tel que l'on a décrit dans cette thèse en particulier dans le chapitre 3.

Les sources utilisées sont l'entrepôt *repository v2* de e-XMLMedia et une base de données relationnelles Oracle 8. L'adaptateur communique avec le *repository v2* en XQuery et récupère un document XML par l'interface XML-DBC

Une meilleur intégration des standards existants a aussi été effectuée. Ainsi, le standard XML-Schema est utilisé pour typer les données et valider les requêtes. Stockés dans une structure tabulaire interne, les XML-Schema permettent de gérer les métadonnées des différentes sources pour la résolution des chemins et la localisation des sources. Les espaces

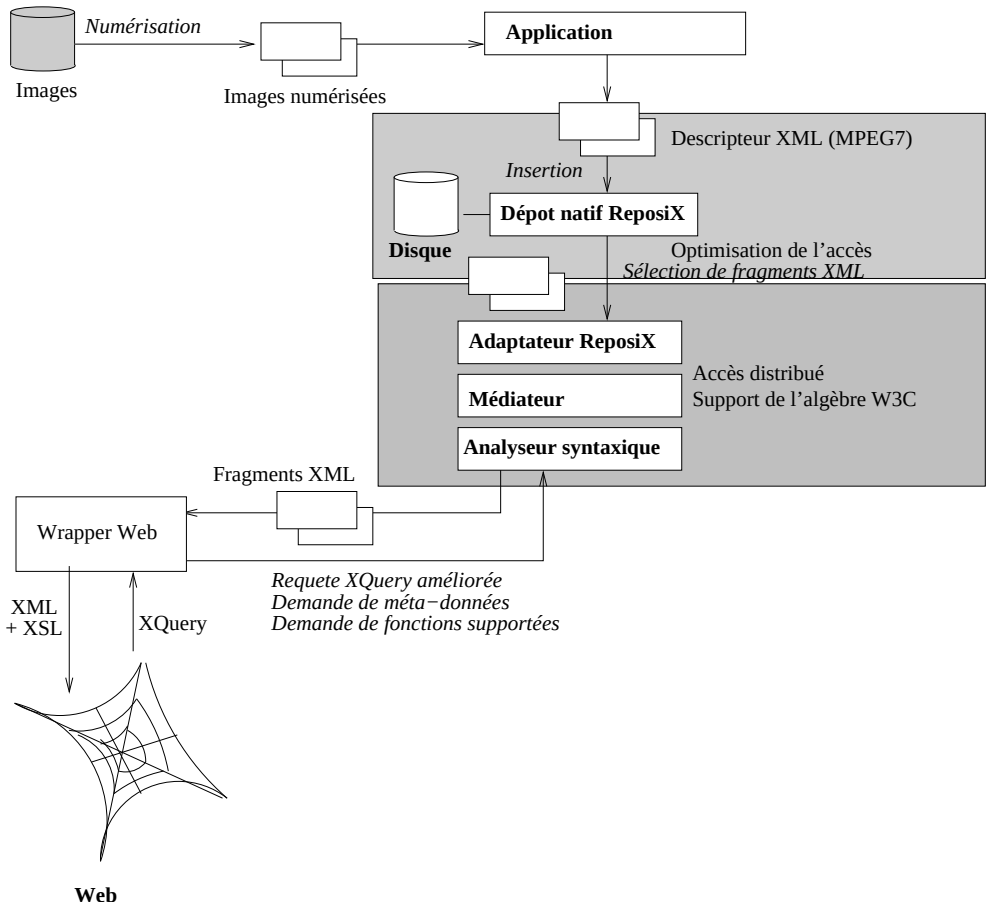


FIG. 7.5 – Architecture du projet MUSE

de noms tels que définis par le W3C ont été entièrement pris en compte. Enfin, le langage de requête XQuery retenu par le W3C comme langage standard pour l'interrogation de données XML a été en grande partie implémenté.

7.6 Conclusion

Les travaux sur les données semi-structurées sont en perpétuelles évolutions et les standards associés sont souvent encore à l'état de proposition, immature ou encore proposés puis rejetés ou améliorés. Ainsi le début de cette thèse en 1998 a coïncidé avec la standardisation du format XML par le W3C. Le modèle de données XML en mémoire (DOM) n'est apparu que vers fin 1998/début 1999. Notre première version du médiateur s'est donc appuyée sur le modèle OEM. Très peu d'outils existants alors sur cette technologie émergente à l'époque, il nous a fallu développer beaucoup d'utilitaires (analyseur XML, schémas navigation d'arbres, représentation graphique d'arbres). Aujourd'hui, on

peut trouver des dizaines de tels outils disponibles publiquement sur l'Internet, alors qu'il nous a fallu beaucoup de temps de développement à l'époque.

Les schémas apparus en 2000/2001 ont représenté un réel bouleversement dans la programmation de notre architecture utilisant jusqu'à ce moment là un modèle bien spécifique assez semblable au Dataguide tout en intégrant le format DTD qui faisait office de standard de schéma à l'époque.

Les plus grands paris lors du développement des premières versions du médiateur ont été le langage de requête et l'algèbre associée qu'il fallait utiliser. Nous nous sommes basé tout d'abord sur une extension de SQL au semi-structuré (que nous avons appelé SQLX), ensuite, pour le projet MIROWEB et jusqu'à la version 1 du médiateur dans le projet XML-KM, nous avons pris le parti d'implémenter le langage XML-QL en suivant au fur et à mesure toutes les versions diffusées sous forme de propositions. Ce n'est qu'en 2001 au moment de la publication du langage XQuery comme standard reconnu, que l'on a mis en chantier le développement de la version 2 du médiateur dans le projet MUSE afin d'utiliser ce langage. Pour cela, nous avons utilisé l'implémentation du langage QUILT que l'on a réécrit pour l'adapter à XQuery.

Dans cette dernière version du médiateur, nous avions fait des choix sur l'algèbre à utiliser, et aucun standard n'étant alors proposé explicitement, nous avons implémenté pour le médiateur une algèbre proche de Niagara sur un modèle de données semblable à $Y_{A}T$. À présent que l'algèbre d'AT&T est proposé comme standard, un nouveau travail pourrait être réalisé.

Chapitre 8

Évaluation

8.1 Introduction

La validation du médiateur se divise en une partie *qualitative* et une partie *quantitative*.

La validation *qualitative* se rapporte aux *cas d'utilisation* (*use-cases*). Il s'agit de montrer en quoi le médiateur *est capable* de répondre aux différentes requêtes qui pourraient lui être posées. Pour cela, un ensemble de cas d'utilisation précis est formulé. Ces cas d'utilisation sont conçus de sorte à couvrir un éventail très large des possibilités qu'un médiateur idéal devrait avoir. Dans le cadre de cette validation, les performances ne sont pas pris en compte. Le seul critère étant si oui ou non, le médiateur est capable dans un contexte précis de répondre à la requête qui lui est posée.

La validation *quantitative* se rapporte aux *bancs d'essai* (*benchmarks*). Il s'agit cette fois-ci de *mesurer* les performances du médiateur dans un contexte donné. Ces performances sont des mesures quantitatives comme le temps d'exécution, la charge mémoire, l'efficacité d'une solution, etc.

Pour valider, nous nous sommes appuyé sur le médiateur v2 en cours de développement à e-XMLMedia. Le médiateur n'intègre pas encore toutes les fonctionnalités décrites dans cette thèse ; en particulier les caches, les modèles de coût, et les langages d'exportation de coût et de capacité. Nous ne pourrons donc pas réaliser d'expérience sur ces derniers points.

Nous validerons tout d'abord qualitativement notre médiateur en reprenant les cas d'utilisation définis par le W3C.

Dans le chapitre 5, nous avons défini des modèles de coût appliqués à une architecture

de médiation telle que présentée dans le chapitre 3. Les modèles de coût qui ont été présentées sont théoriques. L'objectif de ce chapitre est de valider ces modèles de manière expérimentale. C'est ce que nous montrerons dans la validation quantitative.

8.2 Plan du chapitre

Dans la section 8.3, nous verrons en quoi le médiateur *est capable* de répondre aux différentes requêtes qui pourraient lui être posées en l'utilisant sur des cas d'utilisation. Dans la section 8.4, nous montrerons qu'aucun banc d'essai n'est adapté à un médiateur de données hétérogènes. Dans la section 8.5, nous décrirons le système expérimental que nous avons utilisé pour réaliser nos mesures. Dans la section 8.6 nous réaliserons quelques mesures permettant de valider notre prototype. Et enfin nous conclurons dans la section 8.7.

8.3 Cas d'utilisation

Le « *XML Query Working Group* » du W3C a publié une liste de cas d'utilisation (*use-cases*) illustrant les applications du langage XQuery. Ces cas d'utilisation ou *scénarios*, sont regroupé en 9 domaines. Chacun de ces domaines définissant un ensemble de requêtes spécifique à un type d'application. Les domaines sont les suivants :

1. *XMP* : requêtes générales illustrants les besoins des communautés base de données et documentaires.
2. *TREE* : requêtes sur l'extraction d'éléments de documents en préservant les hiérarchies originales.
3. *SEQ* : requêtes basées sur l'ordre dans lequel apparaissent les éléments dans un document.
4. *R* : accès à des base de données relationnelles.
5. *SGML* : transformation de scénarios SGML (un ancien format semi-structuré utilisé en documentaire) en XML.
6. *STRING* : recherche textuelle dans des documents.
7. *NS* : requêtes sur des noms qualifiés dans des espaces de noms.
8. *PARTS* : requêtes récursives sur des documents avec des références externes.
9. *STRONG* : requêtes exploitant des données fortement typées.

Pour chacun de ces domaines, un ensemble de requête couvrant un large éventail des possibilités de XQuery est donné. Pour chacune de ces requêtes, le domaine de définition,

la requête XQuery, sa description « textuelle », et le résultat XML qui devrait être renvoyé comme réponse à la requête si tout se passe bien, sont décrits. Le domaine de définition correspond au contexte dans lequel s'appuie la requête. Il peut s'agir d'un ensemble de documents XML et/ou de DTD.

Le tableau suivant reporte les résultats du médiateur de e-XMLMedia sur les cas d'utilisation du 15 Novembre 2002 recommandé par le W3C (http ://www.w3.org/TR/xmlquery-use-cases/). Les numéros de requêtes mis en gras désignent les requêtes supportées par le médiateur. Les autres (les numéros en italiques), représentent celles qui ne le sont pas.

domaine	description	numéros de requêtes
1 "XMP"	exemples généraux	**1 2 3** *4* **5** *6* **7** *8 9* **10 11** *12*
2 "TREE"	préservant la hiérarchie	**1 2 3 4 5** *6*
3 "SEQ"	basées sur les séquences	*1 2 3 4 5*
4 "R"	accès aux données relationnelles	**1 2 3 4 5 6 7 8** *9* **10 11** *12* **13 14 15** *16* **17 18**
5 "SGML"	basé sur tests SGML	**1 2 3** *4* **5 6** *7 8* **9** *10*
6 "STRING"	recherche de chaînes de caractères	**1 2** *3 4 5 6*
7 "NS"	utilisant les espaces de noms	*1* **2 3 4 5 6** *7* **8**
8 "PARTS"	récursivité et références externes	*1*
9 "STRONG"	utilisant des données fortement typées	*1 2 3 4 5 6 7 8 9 10 11 12*

Le développement du médiateur s'est orienté prioritairement sur l'aspect données plutôt que l'aspect documentation (SEQ, SGML, STRING).

On doit noter pour la validation du support XQuery par le médiateur que ces cas d'utilisation n'est pas suffisant. En effet, la majorité des requêtes de scénario, ne fait intervenir qu'une seule collection donc une seule source. Ce n'est pas satisfaisant pour valider le médiateur. De plus, lorsqu'une seule source intervient dans le traitement d'une requête, le médiateur est optimisé de sorte à passer directement la requête à l'adaptateur correspondant. Cela a pour conséquence que toutes les requêtes ne faisant intervenir qu'une seule source montrent plutôt la capacité à l'adaptateur à répondre à la requête

plutôt que celle du médiateur.

8.4 Bancs d'essai

Un banc d'essai (*benchmark*) est composé d'un ensemble de données et d'une série de requêtes associées à ces données. Les bancs d'essai existants permettent d'évaluer des SGBD relationnels (TCP-C, TCP-H, TPC-D, TPC-R [TPC]), ou objets (OO7 [Carey *et al.* 1993]). Des propositions ont été soumises quand à des bancs d'essai pour des requêtes basées sur des documents XML [Schmidt *et al.* 2001]. On peut retenir l'adaptation de OO7 à XML [Nambiar *et al.* 2001], et le banc d'essai dédié à XML : XMach-1 [Böhme et Rahm 2001].

Mais aucun n'est vraiment adapté à un médiateur de données hétérogènes pouvant aussi bien accéder à des données relationnelles, objets ou semi-structurées. De plus, les requêtes proposées dans ces différents bancs d'essai ne concernent qu'une seule source, et ne présente donc que peu d'intérêt quand aux mesures qui pourraient être faites par rapport à la distribution des données. Nous ne pouvons donc pas nous baser tel quel sur un banc d'essai existant pour valider notre travail.

8.5 Système hétérogène expérimental

Nous nous sommes appuyés sur les données conçues par le générateur du banc d'essai TPC-R. Nous avons ensuite adapté ces données à nos besoins : en effet, TPC-R travaille sur des requêtes en lecture seule sur une seule source basée sur un modèle relationnel. Dans notre cas, il s'agit de travailler sur des requêtes sur des sources réparties.

Le schéma utilisé par TPC-R comporte 8 tables ayant des relations diverses entre elles. Nous avons dans un premier temps généré les données correspondantes à ces huit tables sur environ 1 Gb de données (facteur 1 dans le générateur). Ensuite, nous avons réparti ces 8 tables dans 6 sources autonomes que nous avons défini et nous avons associé les données correspondantes. Trois de ces sources (1, 2 et 3) sont relationnelles et les données n'ont donc subi aucune transformation avant d'y être insérées. Les 3 autres (4, 5 et 6) sont semi-structurées, et afin de pouvoir mettre en valeur cet aspect, les schémas et données ont été modifiés afin de représenter une structure arborescente intéressante avant d'être insérées dans ces dernières.

La figure 8.1 montre les relations entres les tables et la répartition des tables entre les sources. L'annexe C décrit précisément les schémas des tables et les schémas XML utilisés.

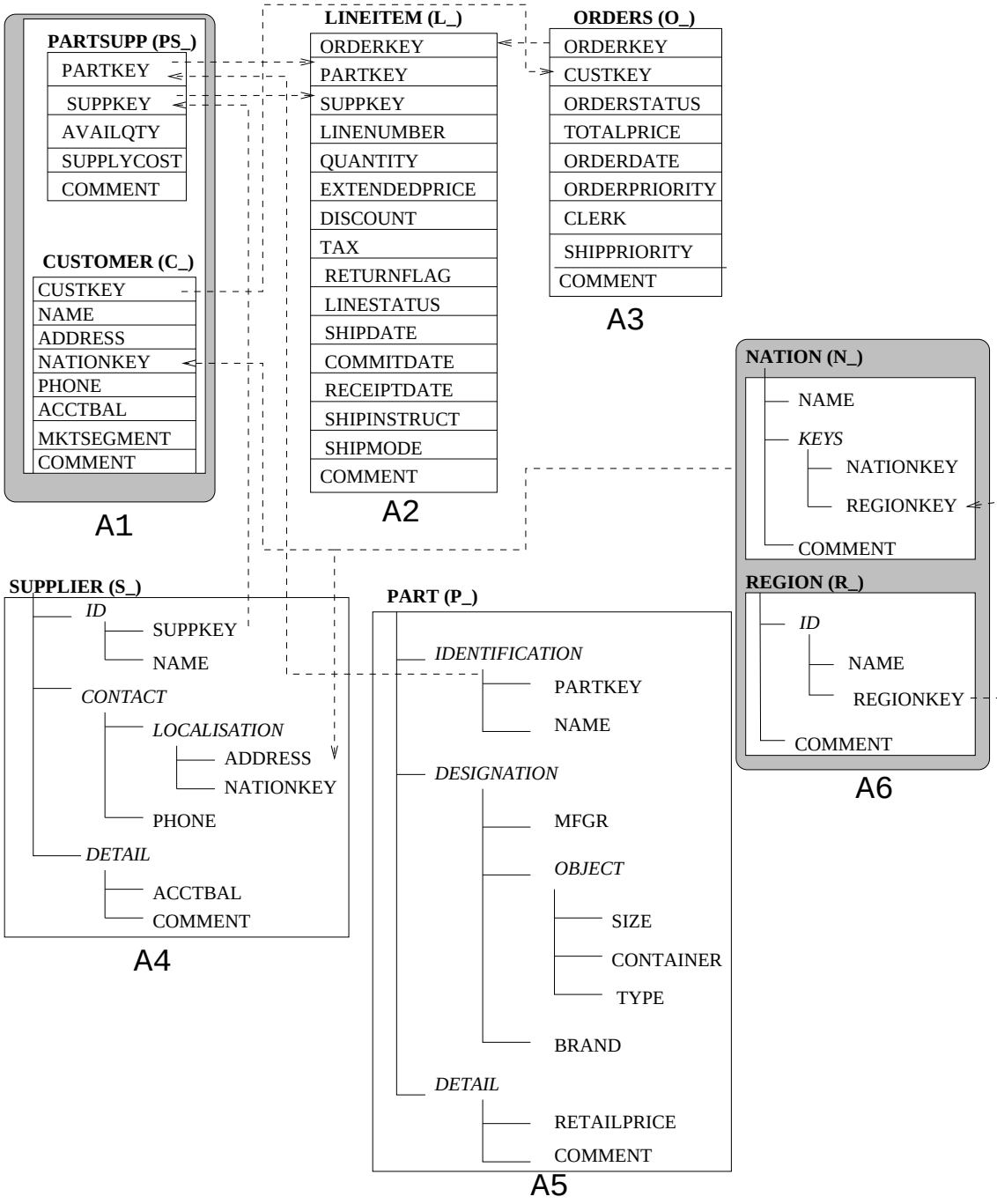


FIG. 8.1 – Schéma de données utilisées pour notre validation

8.5.1 Adaptateur

Les sources 1, 2 et 3 sont trois sources distinctes s'appuyant sur le SGBDR Oracle 8i (version). Elles sont chacune pourvues d'un adaptateur permettant d'interroger une base

de données relationnelles et de renvoyer les résultats sous forme XML.

Dans la suite, SF (*Space Factor*) dépend du coefficient utilisé pour la génération des données.

8.5.1.1 Adaptateur $A1$ pour la source 1

La source 1 comporte deux tables relationnelles :

```
PARTSUPP ( PS_PARTKEY, PS_SUPPKEY, PS_AVAILQTY, PS_SUPPLYCOST, PS_COMMENT )
```

```
CUSTOMER ( C_CUSTKEY, C_NAME, C_ADDRESS, C_NATIONKEY, C_PHONE, C_ACCTBAL,
           C_MKTSEGMENT, C_COMMENT )
```

La table $PARTSUPP$ de la source 1 comporte $||PARTSUPP|| = SF * 800.000$ entrées, et la table $CUSTOMER$ comporte $||CUSTOMER|| = SF * 150.000$ entrées.

8.5.1.2 Adaptateur $A2$ pour la source 2

```
LINEITEM ( L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER, L_QUANTITY,
           L_EXTENDEDPRICE, L_DISCOUNT, L_TAX, L_RETURNFLAG,
           L_LINESTATUS, L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE,
           L_SHIPINSTRUCT, L_SHIPMODE, L_COMMENT )
```

La table $LINEITEM$ de la source 2 comporte $||LINEITEM|| = SF * 6.000.000$ entrées.

8.5.1.3 Adaptateur $A3$ pour la source 3

```
ORDERS ( O_ORDERKEY, O_CUSTKEY, O_ORDERSTATUS, O_TOTALPRICE, O_ORDERDATE,
         O_ORDERPRIORITY, O_CLERK, O_SHIPPRIORITY, O_COMMENT )
```

La table $ORDERS$ de la source 3 comporte $||ORDERS|| = SF * 1.500.000$ entrées.

Les sources 4, 5, 6 sont des sources s'appuyant sur un entrepôt XML non-natif. Elles contiennent les schémas suivants décrits dans l'annexe C.

8.5.1.4 Adaptateur $A4$ pour la source 4

La source 4 comporte $||SUPPLIER|| = SF * 10.000$ entrées et les données sont composées d'arbres de profondeur 4 avec un nombre de fils moyen de 2 par nœud.

8.5.1.5 Adaptateur $A5$ pour la source 5

La source 6 comporte $||PART|| = SF*200.000$ entrées et les données sont composées d'arbres de profondeur 4 avec un nombre de fils moyen de 3 par nœud.

8.5.1.6 Adaptateur $A6$ pour la source 6

La source 6 comporte une collection $||NATION|| = 25$ entrées et les données sont composées d'arbres de profondeur 3 avec un nombre de fils moyen de 2 par nœud ainsi qu'une collection $||REGION|| = 5$ entrées et les données sont composées d'arbres de profondeur 3 avec un nombre de fils moyen de 2 par nœud.

8.5.2 Architecture

Afin de pouvoir classifier différents cas d'intégration de sources, nous définissons plusieurs architecture de médiation. Certaines sont spécifiques à des sources de données relationnelles, d'autres ne traitent que des données semi-structurées, d'autres enfin accèdent directement à toutes les sources, et enfin certaines accèdent à des médiateurs qui eux-mêmes intègrent des sources de données réparties. Nous définissons les différentes architectures de médiation que nous utiliserons tout au long de ce chapitre. Elles sont illustrées à la figure 8.2.

8.5.2.1 Médiateur $M2$ sur données relationnelles

Dans cette architecture (cf. cas (a) de la figure 8.2), nous connectons les adaptateurs $A1, A2, A3$ de sources relationnelles à un médiateur M2.

8.5.2.2 Médiateur $M3$ sur données semi-structurées

Dans cette architecture (cf. cas (b) de la figure 8.2), nous connectons les adaptateurs $A4, A5, A6$ de sources semi-structurées à un médiateur M3.

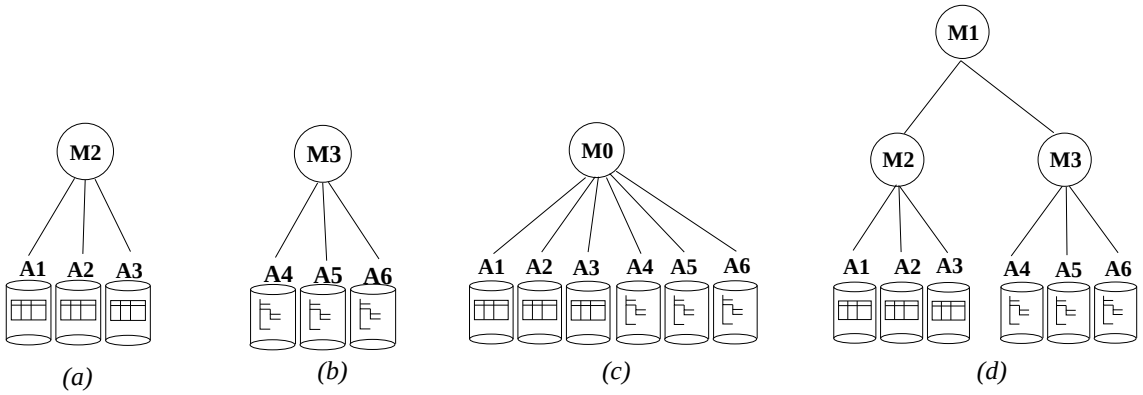


FIG. 8.2 – Architectures d'expérimentation

8.5.2.3 Médiateur $M0$ sur mélange de données relationnelles et semi-structurées

Dans cette architecture (cf. cas (c) de la figure 8.2), nous connectons les adaptateurs $A1, A2, A3, A4, A5, A6$ de toutes les sources, sur un même médiateur M0.

8.5.2.4 Médiateur $M1$ sur médiateurs

Dans cette architecture (cf. cas (d) de la figure 8.2), nous connectons les médiateurs $M2$ et $M3$ définis précédemment à un même médiateur $M1$.

8.5.3 Modèle de données et types de requête

Dans nos architectures de validation, nous faisons appel à des données de type relationnel et à des données de types semi-structurées. Celles-ci sont dispersées sur différentes sources et les attributs de ces différentes données sont générés de sorte à pouvoir réaliser des jointures inter-sites.

8.5.3.1 Modèle de données

Nous nous basons sur le modèle TPC-R avec certaines des tables transformée en document XML comme indiqué à la figure 8.1. Les clefs primaires et étrangères ainsi que les types utilisés pour chaque attribut de chaque table sont décrits dans l'annexe C. Les cardinalités ont été décrites lors de la définition des adaptateurs.

Nous avons généré les données en utilisant SF=0.1 comme facteur de génération. Les primitives exportées par chacun des adaptateurs sont :

```
A1 : primitive Card ("PARTSUPP") := 80000
     primitive Card ("CUSTOMER") := 15000

A2 : primitive Card ("LINEITEM") := 600000

A3 : primitive Card ("ORDERS")   := 150000

A4 : primitive Card ("SUPPLIER") := 20000

A5 : primitive Card ("PART")     := 1000

A6 : primitive Card ("NATION")   := 25
     primitive Card ("REGION")   := 5
```

8.5.3.2 Type de requête

Nous noterons

$$R(A_i, A_j, ...A_n)$$

une requête faisant intervenir les adaptateurs $A_i, A_j, ..., A_n$.

Par exemple $R(A_1, A_2, A_3)$ est une requête faisant intervenir les adaptateurs relationnel A_1, A_2 et A_3. $R(A_1, M_3)$ est une requête faisant intervenir l'adaptateur A_1 et le médiateur M_3 et $R(A_4)$ est une requête s'appliquant simplement à l'adaptateur semi-structuré A_4.

Les requêtes que nous utiliserons pour nos mesures sont des requêtes simples de type SELECT-FROM-WHERE. Nous nous basons sur la structure de requête suivante :

```
for $ps  in collection("PARTSUPP")/*:PARTSUPP
for $c  in collection("CUSTOMER")/*:CUSTOMER
for $l  in collection("LINEITEM")/*:LINEITEM
for $o  in collection("ORDERS")/*:ORDERS
for $s  in collection("SUPPLIER")/*:SUPPLIER
for $p  in collection("PART")/*:PART
for $n  in collection("NATION")/*:NATION
for $r  in collection("REGION")/*:REGION

where

    $ps/PS_PARTKEY = $l/L_PARTKEY
and $ps/PS_PARTKEY = $p/P_PARTKEY
and $ps/PS_SUPPKEY = $l/L_SUPPKEY
and $ps/PS_SUPPKEY = $s/S_ID/S_SUPPKEY

and $c/C_CUSTKEY = $o/O_CUSTKEY
and $c/C_NATIONKEY = $s/S_CONTACT/S_LOCALISATION/S_NATIONKEY
and $c/C_NATIONKEY = $n/N_KEYS/N_NATIONKEY
and $n/N_KEYS/N_NATIONKEY =  $s/S_CONTACT/S_LOCALISATION/S_NATIONKEY

and $r/R_ID/R_REGIONKEY = $n/N_KEYS/N_REGIONKEY

and $o/O_ORDERKEY = $l/L_ORDERKEY

and $c/C_CUSTKEY <  valc
and $o/O_ORDERKEY <  valo
and $s/S_ID/S_SUPPKEY <  vals
and $p/P_IDENTIFICATION/PARTKEY <  valp
and $n/N_KEYS/N_NATIONKEY <  valn
and $r/R_ID/R_REGIONKEY <  valr

and $ps/PS_PARTKEY <  valp
and $ps/PS_SUPPKEY <  vals
and $l/L_ORDERKEY <  valo
and $l/L_SUPPKEY <  vals
and $l/L_PARTKEY <  valp

return

<RESULT>
    <ps>$ps/PS_COMMENT</ps>
    <c>$c/C_COMMENT</c>
    <l>$l/L_COMMENT</l>
    <o>$o/O_COMMENT</o>
    <s>$s/S_COMMENT</s>
    <p>$p/P_COMMENT</p>
    <n>$n/N_COMMENT</n>
    <r>$r/R_COMMENT</r>
</RESULT>
```

où suivant les adaptateurs qu'il faut faire intervenir, nous supprimons les lignes adéquates. De cette façon, nous pouvons tester les jointures inter-sites, et modifier le nombre de réponses obtenues en faisant varier les valeurs de *vali*.

8.5.4 Génération de plans d'exécution

Le générateur de plans d'exécution n'étant pas opérationnel à ce moment, nous rentrons manuellement les différents plans que l'on veut exécuter. Pour cela le plan est décrit sous forme XML et analysé par le médiateur. Le document XML décrivant le plan est une simple transcription de l'arbre algébrique que l'on veut exécuter. On y décrit les différentes sources, les opérateurs et les expressions à utiliser.

Chaque opérateur de l'arbre algébrique du plan d'exécution (voir sous-section 3.7.4 du chapitre 3) a comme propriétés :

- *l'expression associée (expression)* : il s'agit de l'expression XQuery associée à cet opérateur ;
- *les chemins (path)* : il s'agit de l'ensemble des chemins que l'opérateur devra récupérer dans les XTuples de sorties.

À chacun des opérateurs, est associé un élément XML auquel les propriétés décrites sont encapsulées.

Soit le plan d'exécution donné à la figure 8.3.

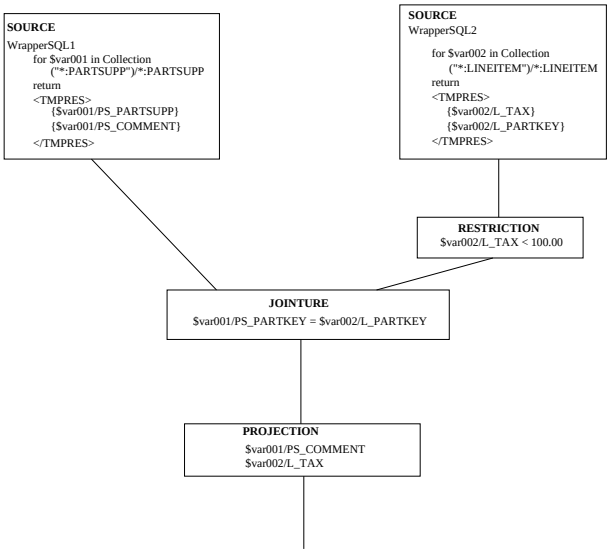


FIG. 8.3 – Plan d'exécution

Ce plan d'exécution peut être représenté par le document XML suivant.

```
<plan>
  <projection>
    <paths>
      <path>$var001/PS_COMMENT</path>
      <path>$var002/L_TAX</path>
    </paths>
    <join>
      <expression>
        $var001/PS_PARTKEY = $var002/L_PARTKEY
      </expression>
      <source name="WrapperSQL1">
        <expression>
          for $var001 in collection("*:PARTSUPP")/*:PARTSUPP
          return
            <TMPRES>
              { $var001/PS_COMMENT  }
              { $var001/PS_PARTKEY }
            </TMPRES>
        </expression>
        <paths>
          <path>$var001/PS_PARTKEY</path>
          <path>$var001/PS_COMMENT</path>
        </paths>
      </source>
        <restrict>
          <expression>
            $var002/L_TAX < 100.00
          </expression>
          <source name="WrapperSQL2">
            <expression>
              for $var002 in collection("*:LINEITEM")/*:LINEITEM
              return
                <TMPRES>
                  { $var002/L_TAX  }
                  { $var002/L_PARTKEY }
                </TMPRES>
            </expression>
            <paths>
              <path>$var002/L_TAX</path>
              <path>$var002/L_PARTKEY</path>
            </paths>
          </source>
        </restrict>
    </join>
  </projection>
</plan>
```

Cette méthode de représentation de plan d'exécution permet de construire des plans d'exécution particulier afin de réaliser des tests.

8.6 Expérimentation

Les données sont stockées sur des bases de données Oracle 8i tournant sur un processeur Pentium 4 1,6 GHz avec 256 Mb de RAM sous Linux 2.4.20. Le médiateur est installé sur un Pentium Celeron 600Mhz avec 64M de RAM sous Linux 2.4.2 connecté aux sources de données par une liaison à 10Mb/s.

Pour les mesures, nous avons utilisés la méthode Java $System.currentTimeMillis()$. Comme cette méthode donne le temps réel et nom le temps système et utilisateur, elle est très dépendante de l'environnement à un temps donné (multi-tâche). C'est pour cela qu'il faut prendre les temps en considérant leur rapport plutôt que par rapport à leur valeur absolue.

8.6.1 Surcoût induit par l'architecture de médiation

Nous voulons montrer le surcoût induit par les temps de communication entre les différents médiateurs. Pour cela, nous lançons une même requête $R(A_3)$ sur l'architecture M0 (accès direct du médiateur aux données) puis sur l'architecture M1 (arborescence de médiateur). Le temps mesuré en millisecondes en fonction du nombre de nœuds résultats est représenté pour une architecture de type M1 et une architecture de type M0 sur la figure 8.4.

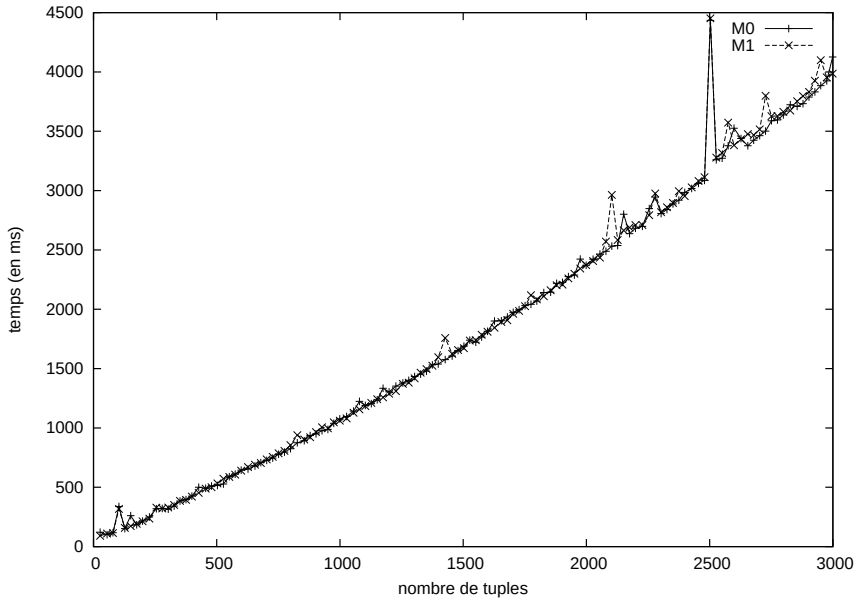


FIG. 8.4 – Temps de réponse suivant l'architecture M0 et M1

Nous constatons que les courbes sont très proches. « Empiler » des médiateurs ne

nuit pas à l'évaluation. Dans notre expérience, les deux médiateurs se trouvent sur la même machine. D'après le résultat de notre expérience, nous pouvons en déduire que si les médiateurs se trouvaient sur des machines distinctes, seuls le temps de communication réseau serait à prendre en compte dans le surplus de temps.

8.6.1.1 Temps de chaque phase

L'exécution d'une requête se décompose suivant les phases suivantes :

1. *analyse de la requête* : transformation de la requête XQuery en un modèle de donnée de requête interne ;
2. *construction de l'arbre algébrique* : normalisation, canonisation et atomisation. Construction de l'arbre de dépendance, et enfin construction de l'arbre algébrique ;
3. *initialisation de l'exécution* : connexion à l'adaptateur et récupération du premier XTuple ;
4. *exécution de la requête sur l'adaptateur* : envoie de la requête à l'adaptateur, récupération du résultat par XML/DBC sous forme de flux SAX. Transformation du flux SAX en XTuple ;
5. *exécution de la requête et recomposition* : passage des XTuples à travers tout l'arbre algébrique pour évaluer le résultat.

   Les phases 1, 2 et 3 constituent la phase d'initialisation de la requête

La phase d'initialisation, et les phases 4, 5 ainsi que le temps total est représenté sur la figure 8.5.

On voit sur la courbe que la phase d'initialisation est quasi-négligeable par rapport au temps d'exécution de la requête sur l'adaptateur et le temps d'exécution sur le médiateur. L'adaptateur se situe sur la même machine que le médiateur. Comme il s'agit de tester le temps passé par une requête dans les différentes phases, une requête simple mono-source $R(A_3)$ a été prise :

```
for $o  in collection("ORDERS")
where
   $o/O_ORDERKEY < valo
return
   <RESULT>
      <o>$o/O_COMMENT</o>
   </RESULT>
```

où la valeur de `valo` est changée suivant les requêtes.

La courbe 8.6 trace les temps des phases 1, 2 et 3 de la phase d'initialisation.

Le temps d'analyse de la requête est assez petit. La phase de génération du plan prend environ le double de temps. Enfin la phase d'initialisation, c'est à dire le temps de

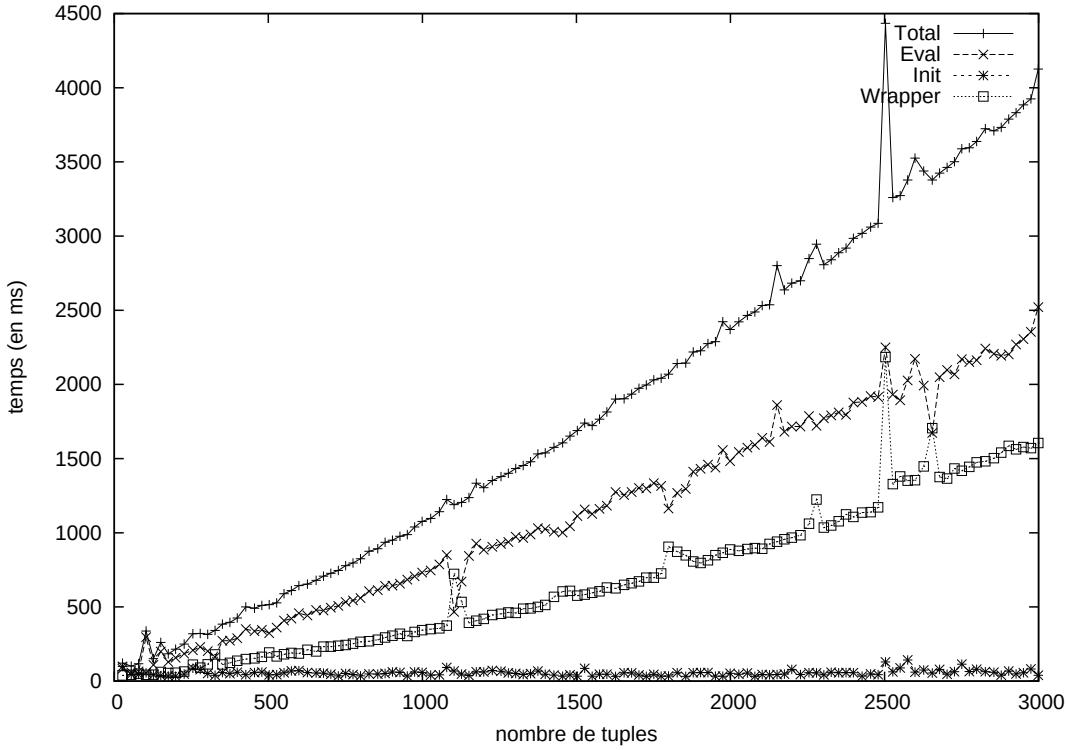


FIG. 8.5 – Temps des différentes phases

connexion aux adaptateurs ainsi que d'initialisation du premier résultat est un peu plus élevé.

Pour information, la phase d'initialisation du médiateur est de 91570 ms dont 91118 ms pour la récupération des métadonnées. La phase de déconnexion du médiateur est de 280 ms. Ces dernières aux contraires des phases citées pour l'exécution d'une requête ne sont effectuées qu'une seule fois lors du lancement du médiateur et de son extinction.

8.6.2 Coût de la reconstruction

Afin de pouvoir évaluer le coût de la reconstruction, le plus simple est de soumettre une requête très simple à une seule source de données relationnelles. L'évaluation est alors réduite à trois étapes :

1. Transformation de la requête XQuery en SQL.
2. Si l'on considère que la base est déjà connectée, exécution de la requête sur la base de données.
3. Récupération des tuples, et transformation des tuples résultat en XTuples.

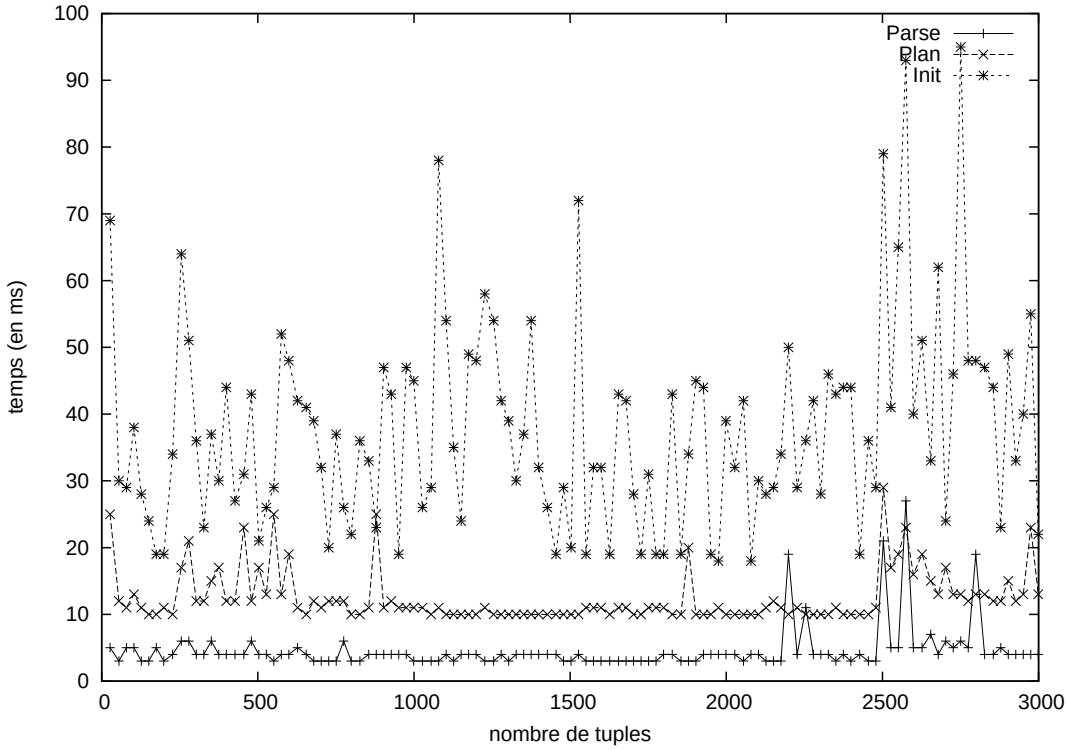


FIG. 8.6 – Décomposition de la phase d'initialisation

L'étape 1 est quasi-immédiate, l'étape 2 a pour coût le coût d'exécution normal d'une requête SQL sur la source de données. La durée de l'étape 3 correspond à la récupération des données par le médiateur et à la restructuration. Dans le cas d'une source unique, le coût est directement lié au nombre de tuple qui sont retournés. La requête $R(A_3)$ de test est :

```
for $o in collection("ORDERS")/*:ORDERS
where
    $o/O_ORDERKEY <  X
return
<RESULT>
    <o>$o/O_COMMENT</o>
</RESULT>
```

Nous faisons varier X entre 0 et 3000. La table $ORDERS$ a été remplie avec 3000 tuples dont la clef unique $O\_ORDERKEY$ varie de 1 à 3000. Les temps des différentes étapes de l'exécution sont donnés dans la courbe 8.7

Les temps de l'étape 1 sont approximativement constants et inférieur à 10 ms.

Si l'on considère le surplus de temps, l'étape 1 est constante et non significative, l'étape 2 est commune à une requête SQL « normale » et une requête XQuery, l'étape 3

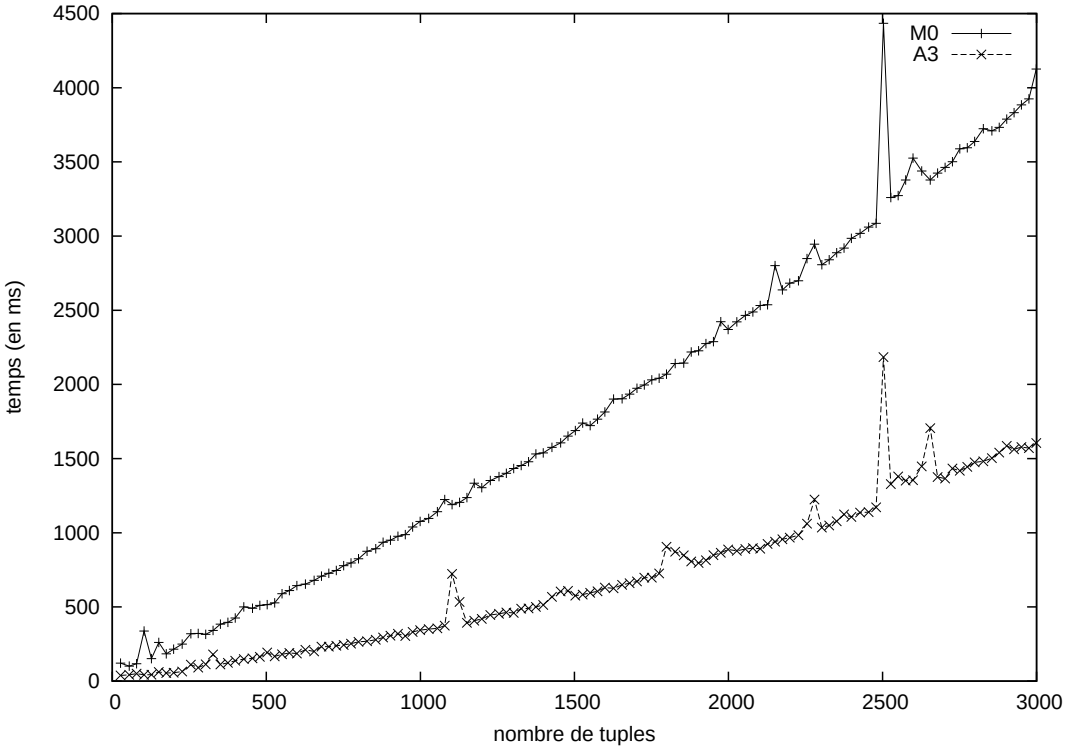


FIG. 8.7 – Surplus de temps dû à l'architecture de médiation

est la plus coûteuse. Dans l'exemple, les règles de reconstruction est basique, mais faire des reconstructions plus complexes avec de nombreux nœuds intermédiaires fait évidemment grossir la structure.

Si l'on considère le rapport des temps mis par l'architecture M0 et par l'architecture s'adressant directement à A3 (figure 8.8), on constate effectivement que ce rapport est borné.

8.6.3 Jointures inter-sites

Nous soumettons une requête $R(A_2, A_3)$ faisant intervenir une jointure entre plusieurs tables.

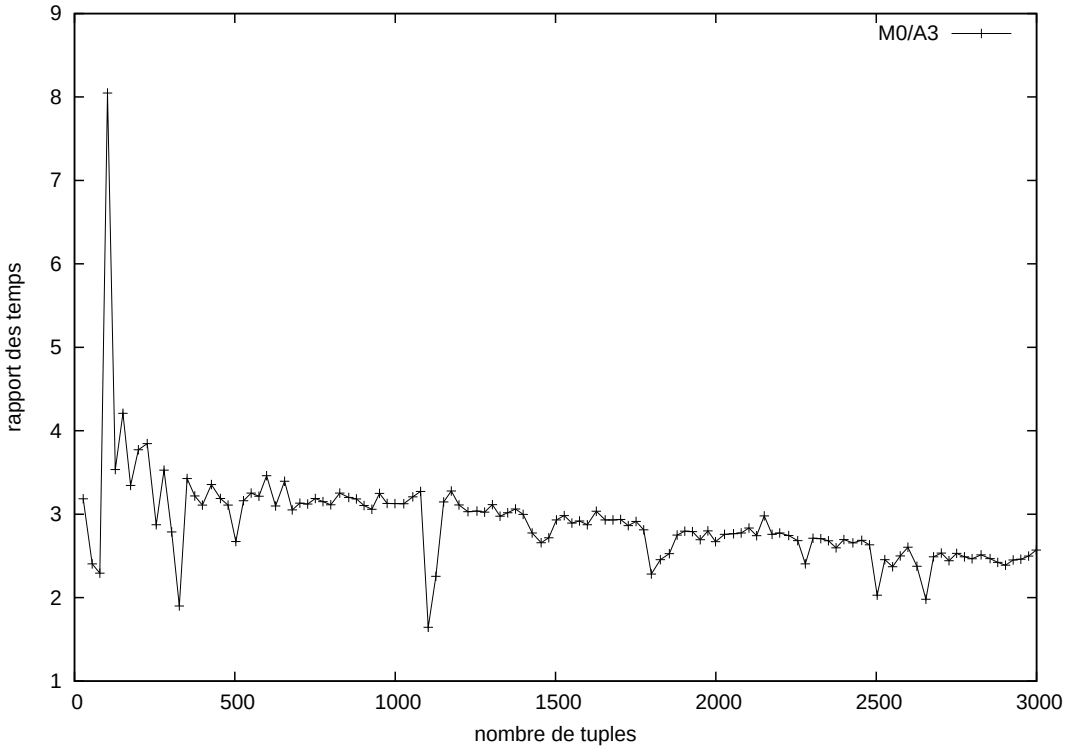


FIG. 8.8 – Rapport du temps mis par M0 et par A3

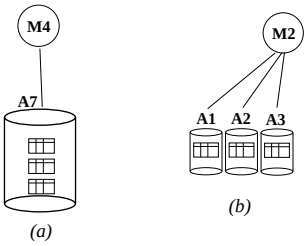


FIG. 8.9 – Architecture M2 et M4

```
for $l  in collection("LINEITEM")
for $o  in collection("ORDERS")
where
        $o/O_ORDERKEY = $l/L_ORDERKEY
and      $l/O_ORDERKEY < valo
return
<RESULT>
        <l>$l/L_COMMENT</l>
        <o>$o/O_COMMENT</o>
</RESULT>
```

Nous évaluons cette requête une fois en adoptant l'architecture M2 et une fois en adoptant l'architecture M4. Dans le premier cas, les tables se trouvent toutes sur le même

site. La jointure est donc réalisée par la source elle-même. Dans le second cas, les tables sont réparties sur différents sites. La jointure est donc réalisée au niveau du médiateur.

Le résultat des expériences est représenté à la figure 8.10.

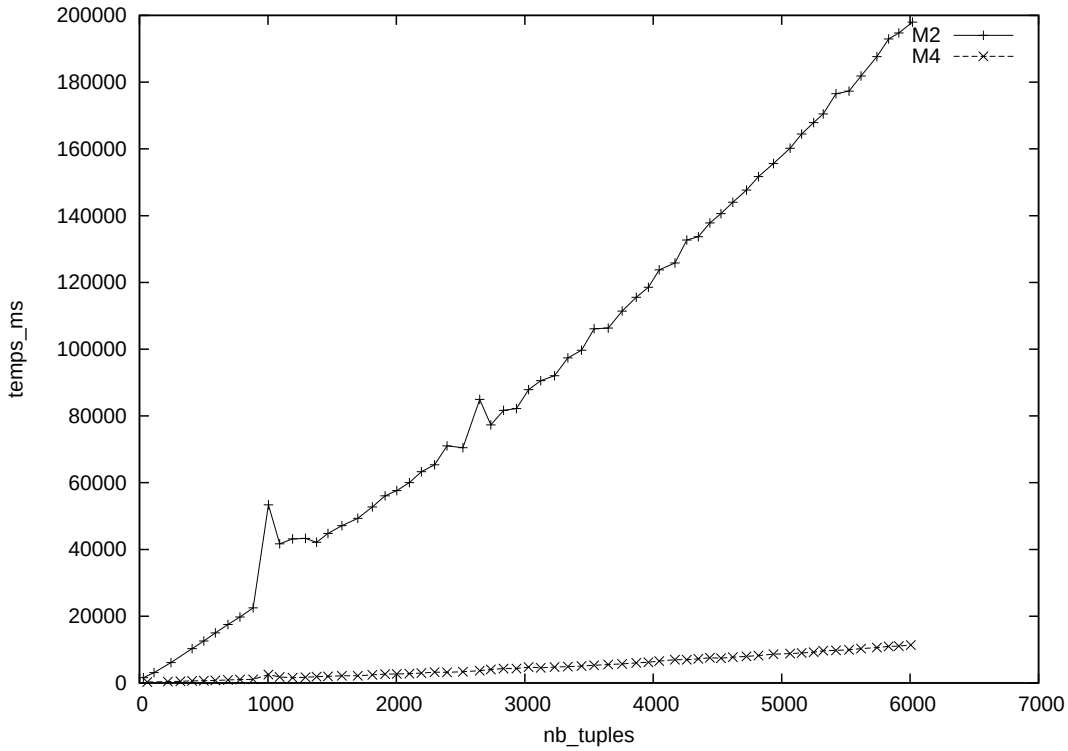


FIG. 8.10 – Comparaison architecture M2 et M4

Les jointures intersites sont des opérations très coûteuses. Le rapport des deux courbes (figure 8.11) montre que ce rapport est de l'ordre de 20 environ. Néanmoins, ce rapport est borné et semble décroître.

8.7 Conclusion

Parmi les cas d'utilisation proposés par le W3C, tous ceux spécifiques à l'orientation actuelle du médiateur sont évalués correctement. Notre objectif premier ayant été d'intégrer un SGBD relationnel avec des données XML orienté données (*data centric*) d'un entrepôt XML, ce sont les cas d'utilisation faisant intervenir des requêtes concernant ces préoccupations qui ont été privilégiés. Afin de pouvoir achever l'exécution correcte de tous les cas d'utilisation, il manque encore à se préoccuper des requêtes sur des données XML orientées document.

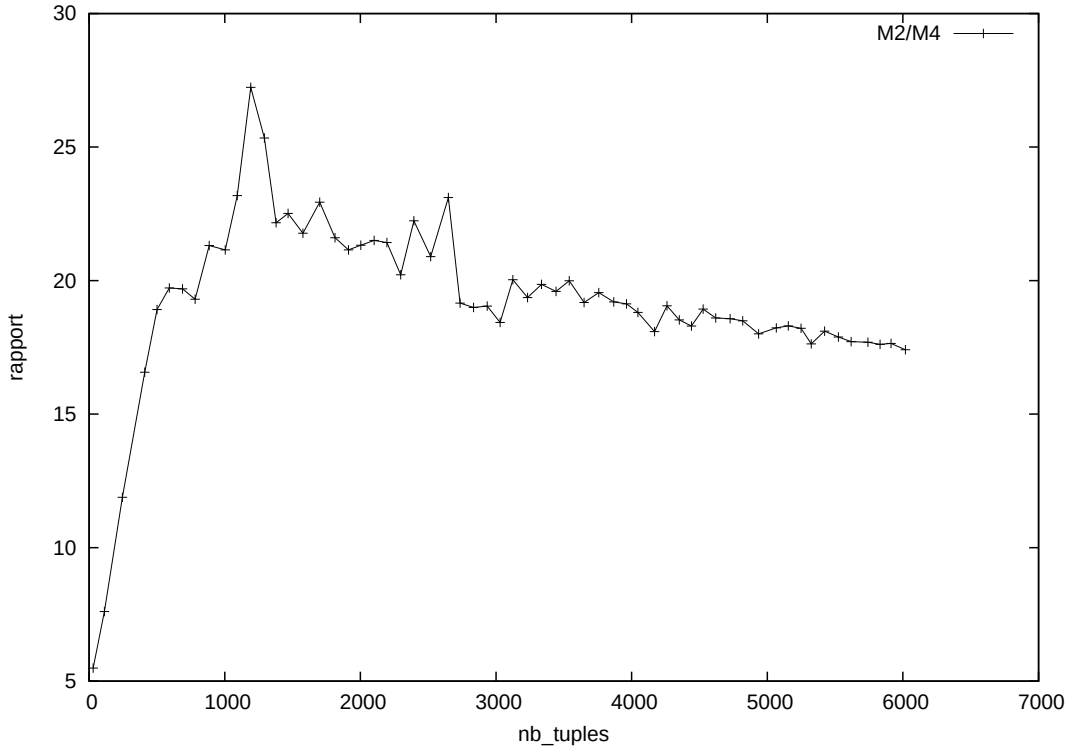


FIG. 8.11 – Rapport d'architecture M2 et M4

Les bancs d'essai permettant d'évaluer les performances d'un médiateur XML ne sont pas encore véritablement définis. Parmi les bancs d'essai existants, on peut citer trois catégories. Des bancs d'essai trop orientés documents ne sont pas utilisables, car notre médiateur n'intègre pas encore toutes les fonctionnalités pour le faire. Des bancs d'essai concernant des données XML sur une seule source dans ce cas, les performances d'évaluation du médiateur n'ont que peu de sens. Enfin, des bancs d'essai font intervenir la distribution de données sur différentes sources, mais qui ne sont pas dans un contexte semi-structuré.

C'est pour cela que nous avons réalisé des tests sur le modèle de données du banc d'essai TPC-R auquel nous avons apporté deux modifications. La première modification a consisté à modifier les tables et les données afin d'avoir des de données relationnelles et des données XML pour pouvoir tester l'intégration de données hétérogènes. La seconde modification a consisté à répartir ces données sur des sources pouvant se trouver sur différents sites, ceci afin de tester la distribution des données.

Les requêtes ont été effectuées afin de vérifier des comportements spécifiques du médiateurs. Ainsi, des expériences ont été effectuées afin d'évaluer le surcoût d'une architecture de médiation, d'abord par rapport à une source simple, puis par rapport à un arbre de médiation.

D'autres expériences ont servi à évaluer le temps passé dans chacun des modules de l'architecture de médiation.

Chapitre 9

Conclusion

Dans cette thèse, nous nous sommes intéressés au problème de l'optimisation de requête pour données semi-structurées dans une architecture hétérogène. Nous avons défini le cadre général de l'optimisation, ainsi que des règles d'optimisation et de calcul de modèles de coûts adaptés au contexte. Nous avons basé notre étude sur le système de médiation de bases de données hétérogènes développé à la société e-XMLMedia.

Le cadre général de la médiation a été défini en répondant à un certain nombre de questions de base. Nous avons étudié les caractéristiques d'une architecture de médiation faisant intervenir des données semi-structurées comme modèle de données interne. Nous avons montré les différents concepts liés aux données semi-structurées et les relations qui existent entre eux, ainsi que les difficultés que présentait l'évolution des travaux existants du relationnel et orientés objets vers le semi-structuré.

Nous avons aussi étudié les mécanismes permettant d'évaluer une requête au sein du moteur d'intégration. Pour cela, nous avons conçu une algèbre permettant de modéliser les opérations à effectuer pour répondre à une requête donnée ainsi que la manière d'évaluer cette algèbre. Partant du fait que l'évaluation d'une requête doit être la plus efficace possible, nous nous sommes penché sur les différentes manières d'optimiser l'exécution. Nous avons tout d'abord cherché à estimer le coût d'exécution des requêtes en proposant un modèle de coût. Ensuite, nous avons suggéré l'utilisation d'un cache sémantique s'appuyant sur un SGBD natif XML pour conserver des résultats de requêtes antérieures dans le but de les réutiliser ultérieurement afin d'économiser des coûts de communication.

Enfin nous avons validé nos résultats lors de l'implémentation de prototypes dans des projets de recherche. Parmi les cas d'utilisation proposés par le W3C, tout ceux spécifiques à l'orientation actuelle du médiateur sont évalués correctement (requêtes sur SGBD relationnels, médiation de différentes sources). Constatant qu'il n'existe pas de banc d'essai dédié à la médiation de données semi-structurées, nous avons adapté et étendu le banc d'essai TPC-R à ce contexte. Les premières mesures de performance sont

encourageantes.

9.1 Résumé des contributions

Pour tenter de résoudre de façon plus satisfaisante les problèmes liés à l'intégration de données semi-structurées, nous avons proposé des solutions originales avec les contributions suivantes.

Spécifications d'une architecture « tout-XML ». L'originalité de notre approche est de s'appuyer entièrement sur les standards XML pour bâtir une architecture modulaire. Plutôt que de re-inventer de nouveaux concepts à chaque problème, nous avons préféré étendre et adapter des solutions existantes. Ainsi, le médiateur respecte et s'appuie sur les normes XML, y compris XML-Schéma, XQuery, les interfaces DOM et SAX. Les XML-Schemas sont intensivement employés pour la représentation de métadonnées. En particulier, ils décrivent les sources et les vues de données situées sur n'importe quelle couche. Le typage d'une requête XQuery est vérifié à l'aide des schémas. Nous supportons actuellement la plupart des cas d'utilisation de XQuery. Nous traitons XML de façon interne sous forme de flux d'événements SAX pour des raisons d'efficacité. En effet, DOM est en général trop coûteux pour instancier des documents XML pendant le traitement. Cependant, l'utilisateur peut s'il y a lieu obtenir des arbres DOM comme résultat et nous employons parfois DOM à l'intérieur du médiateur pour conserver des documents XML pour des traitements futurs.

Une autre originalité est de définir des formats de communication d'information entre les adaptateurs et le médiateur entièrement en XML. Ainsi, les informations de schémas, de coûts et de fonctionnalités sont envoyées sous forme de fichiers XML. Les requêtes et les résultats sont exprimés en XQuery et XML respectivement. Ces formats de communication s'appuient sur une interface applicative de programmation nommée XML/DBC qui a été définie par la sociéte e-XMLMedia pour gérer des sources XML.

Tout ceci contribue à l'architecture « Tout-XML » que nous avons définie dans cette thèse.

Une algèbre adaptée aux données semi-structurées et une évaluation en flux. Nous avons proposé une nouvelle algèbre que l'on a nommé XAlgèbre pour manipuler des données XML. Cette algèbre est une extension de l'algèbre relationnelle. Notre but est d'être aussi près que possible de l'algèbre relationnelle étendue [Zaniolo 1985], tout en permettant la manipulation efficace des arbres et des collections ordonnées d'arbres.

À l'instar des relations en bases de données relationnelles, nous introduisons les

XRelations. Avec les XRelations, les domaines sont des arbres XML construit sur un ensemble donné de chemins (le guide de données). Les attributs sont des XPath mettant en référence des noeuds dans les arbres XML. Chaque attribut peut être multi-valué, c'est-à-dire référence plusieurs sous-arbres. Les XRelations sont des collections ordonnées de XTuples. Ainsi, chaque XTuple se compose d'attributs appelés XPath, dont les valeurs mettent en référence des sous-arbres dans la collection d'arbres.

La XAlgèbre inclut les opérations relationnelles permettant de traiter à la fois les tables, les références et la navigation dans les arbres XML. L'algèbre est une algèbre physique dans le sens où les expressions algébriques sont employées pour traiter des flux XML et que les algorithmes mettent directement en œuvre les opérateurs. Les documents XML sont envoyés au médiateur sous forme de flux d'événements (basés sur SAX). Les XTuples sont créés "au vol" quand les documents XML de schémas connus sont reçus des adaptateurs. Les opérateurs non bloquants travaillent en pipeline sur le flux d'événement. Les opérateurs bloquants exigent une instanciation complète d'un flux d'entrée dans le cache. Les opérateurs n-aires non bloquants peuvent en général travailler en parallèle sur les flux d'entrée. Tous les opérateurs de la XAlgebra reçoivent une collection de XTuples en entrée et renvoient une collection de XTuples en sortie. En général, nous modifions directement la XRelation en mémoire. Les opérateurs ont également des paramètres spécifiques. Le processus d'évaluation de chaque opérateur se compose de deux étapes : une étape de préparation et une étape d'exécution. L'étape de préparation analyse le(s) XRelation(s) d'entrée et le(s) paramètre(s) associé(s) à l'opérateur pour déterminer quelle sera l'opération exacte à effectuer quand les XTuples arriveront. Par exemple, pour une opération qui exige de fusionner des arbres, l'étape de préparation détermine à quel nœud de référence le nouveau sous-arbre devra être lié et quels chemins seront mis en commun. Ainsi, l'étape d'exécution est efficace, puisque la majeure partie du traitement a déjà été faite.

Une extension du modèle de modèle de coût d'une architecture de médiation au semi-structuré. La solution classique pour choisir le meilleur plan d'exécution est de comparer les coûts des différents plans en utilisant un modèle de coût. Nous proposons un modèle de coût fortement inspiré de DISCO [Naacke *et al.* 1998]. Le médiateur est muni d'un modèle de coût générique dérivé d'un modèle de coût relationnel étendu avec la manipulation d'arbre. Chaque adaptateur peut alors exporter des statistiques et des formules détaillées de coût vers le médiateur. Le modèle générique de coût est généralement employé avec des exportations de statistiques (pour évaluer des cardinalités), et les formules spécifiques exportées par un adaptateur peuvent surcharger les formules génériques. Cette approche donne un cadre pour calculer le coût global d'un plan de requête intégrant l'information locale des sources. Pour communiquer leur modèle de coût au médiateur, un adaptateur emploie un langage de modèle de coût. Dans un environnement XML tel que nous l'avons défini, le langage de coût est évidemment défini en XML. Comme les formules et les définitions de statistiques emploient beaucoup de notations mathématiques, nous avons construit une proposition de langage de coût sur MathML, une spécification du W3C pour coder en XML la représentation ou la structure d'un objet mathématique. Les avan-

tages d'employer le format MathML pour décrire des formules de coût sont triples : il est entièrement en XML, il supporte des formules générales, et des logiciels courants de calcul peuvent être employés pour calculer les formules. Les paramètres utilisés pour l'évaluation d'un modèle de coût sont des statistiques relatives au système (statistiques système) et des statistiques relatives aux données (statistiques de données). Pour des données semi-structurées, quelques autres paramètres système sont définis, comme la comparaison entre deux valeurs typées, la comparaison entre deux arbres, la navigation dans un arbre (suivi de pointeurs). Une ou plusieurs formules sont définies afin de calculer le coût d'évaluation d'une requête dans ce système (grosse granularité) ou un attribut dans un opérateur particulier (granularité fine). Les formules pour les granularités les plus fines sont spécifiques aux sources et peuvent être exprimées avec des paramètres spécifiques. Les formules pour les granularités les plus grandes se composent de la cardinalité, du coût total et du coût d'exécution.

Un langage permettant d'exporter des capacités. Dans la version décrite du médiateur, les capacités de source sont prises en considération par catégorie. Nous supportons trois catégories des sources : des sources XQuery, des sources SQL, et des fichiers XML. Fondamentalement nous poussons les requêtes XQuery aux sources XQuery, les requêtes SQL de base aux sources SQL, et les sélections aux fichiers gérés par un filtre. Cela est insuffisant pour prendre en compte des capacités de traitement détaillées des sources. Pour aller plus loin et tenir compte des capacités de traitement détaillées des sources au niveau du médiateur, une description précise des capacités de traitement de l'adaptateur est exigée. Ceci peut être fait globalement pour un adaptateur en envoyant un message d'information détaillant quels opérateurs XML sont autorisés sur toutes les collections ou spécifiquement sur une collection. Les règles les plus spécifiques prévalant toujours.

Nous avons proposé un langage de communication de capacité entièrement basé sur XML permettant aux adaptateurs de communiquer leurs possibilités d'interrogation au médiateur.

L'adaptation d'un banc d'essai TPC-R pour évaluer des requêtes sur XML distribués. Afin de pouvoir évaluer les performances de notre architecture de médiation, nous avons proposé une extension du banc d'essai TPC-R à un contexte de médiation de données semi-structurées.

L'utilisation d'un cache sémantique basé sur un SGBD natif XML. Nous avons aussi proposé dans cette thèse une manière de réduire la transmission de messages en mettant en œuvre un cache sémantique au niveau du médiateur. Les résultats XML répondant à une requête exécutée sur le médiateur peuvent être conservés dans un tel cache. Pour conserver des documents XML et y accéder de la façon la plus efficace possible, nous

avons introduit une architecture d'entrepôt natif en décrivant ReposiX qui a été conçu et développé pour le projet MUSE au laboratoire PRiSM. Une exploration sur la façon d'utiliser cet entrepôt natif comme cache sémantique a été proposée.

Ainsi une table des requêtes exécutée ordonnée par horodate d'exécution avec résultats associés est maintenue dans le cache. Celle-ci serait employée pour répondre à de nouvelles requêtes. Naturellement, la mise à jour sur les données des sources ne peut être prise en considération sans mécanisme de remontée d'événements. Ainsi, le cache sémantique est seulement utilisable sous certaines collections de documents XML non mis à jour fréquemment. Il est cependant de grande utilité dans le cas de sources lentes, par exemple, les sources web. Avec le cache sémantique, une nouvelle requête devrait d'abord être vérifiée par le cache pour déterminer si il peut répondre totalement ou en partie à la requête. Si oui, la requête est divisée en deux parties (une partie peut être nulle) : une requête locale qui peut être exécutée par le cache et une requête de source qui doit être exécutée par les sources distantes. Les deux résultats doivent être correctement assemblés. Ceci peut être fait en comparant les formes canoniques des arbres algébriques associés à la requête à celle de chaque requête du cache. Si l'une est un sous-ensemble de l'autre, le cache peut être employée pour traiter une partie de la requête. L'arbre algébrique de requête doit être élagué pour remplacer la partie commune par un appel aux XRelations du cache. Employer un cache sémantique XML pour XQuery est un sujet complexe qui doit être encore étudié, mais qui pourrait être très bénéfique aux performances.

9.2 Travaux à court terme

À court terme, nous pensons étendre le travail que nous avons réalisé dans cette thèse :

- nous envisageons tout d'abord de terminer l'implémentation de toutes les règles d'optimisation et de génération de plans d'exécution ainsi que de compléter et implémenter le langage d'exportation de formules de coûts et de statistiques que nous avons définis ;
- nous allons ensuite terminer l'entrepôt natif ReposiX et le connecter au médiateur en tant que cache ;
- nous devrons ensuite permettre une approche plus fine des capacités des sources, et à l'instar de GARLIC ou de DISCO, définir un langage permettant aux sources de décrire les fonctionnalités dont elles sont capables ;
- enfin, une fois notre optimiseur complètement achevé, nous pourrons réaliser des tests sur le choix du meilleur plan d'exécution parmi l'ensemble des plans générés. Nous n'avons pas pu le faire dans cette thèse car la version du médiateur intégrant la génération automatique des plans et le calcul des modèles de coût n'est pas disponible dans son ensemble.

9.3 Travaux de recherche futurs

En plus de l'approfondissement des travaux que nous venons d'exposer. Nous pensons par la suite nous consacrer à un certain nombre de recherche qui permettront de compléter utilement ces travaux.

9.3.1 Optimisation des plans d'exécution

Un axe de recherche porte sur l'optimisation des plans d'exécution pour l'évaluation des requêtes.

Formule d'équivalence de la XAlgèbre Un des objectifs est d'approfondir la XAlgèbre en trouvant des propriétés, des règles d'équivalence et en complétant cette XAlgèbre afin qu'elle soit la plus complète possible

Générateur de plan d'exécution A partir des règles d'équivalences on pourra à partir d'un plan d'exécution, créer un générateur de plans d'exécution équivalent (c'est-à-dire donnant les mêmes résultats à l'exécution).

Algorithmes utilisés par les opérateurs La version testée du médiateur utilise un algorithme simple de jointure (boucles imbriquées optimisées). Il est évident que d'autres algorithmes devraient être considérés, pour la jointure notamment, mais aussi pour d'autres opérateurs (par exemple, pour l'imbrication qui est assez complexe). Implémenter la jointure dépendante, c'est-à-dire, une jointure lisant une XRelation et en interrogeant l'autre avec la valeur lue, pourrait être utile pour gagner en nombre des messages en cas de résultat de faible cardinalité. La jointure par tri-fusion et la jointure par hachage pourraient également être utiles. Ainsi, nous aurons actuellement une bibliothèque d'algorithmes pour chaque opérateur de la XAlgebra. Le problème sera alors de choisir le meilleur plan.

Requêtes paramétrées Les requêtes XQuery sont compilées dans des plans d'exécution exprimés en algèbre relationnelle étendue capable de traiter les arbres XML en pipeline. Le traitement des requêtes est clairement divisé en étapes. Nous avons isolé l'étape de réécriture de requête de l'étape de décomposition qui produit des arbres algébriques traitant des sources de données localisées. La localisation des collections est effectuée en utilisant les métadonnées sous la forme de schémas XML. Nous voudrions également séparer plus clairement la phase compilation de requêtes éventuellement paramétrées de la phase

exécution. Il serait aussi possible de développer une X-machine virtuelle distribuée plus efficace pour traiter des expressions de XAlgebra sur des flux XML.

Prise en charge des capacités des adaptateurs Il faudra aussi pouvoir intégrer dans les calculs de coût et dans le générateur de plans d'exécution, la prise en charge des opérations non traitables par les adaptateurs.

9.3.2 Modules d'optimisation

Un autre axe de recherche porte sur des modules qui pourraient être ajoutés dans le but d'optimiser les performances globales du médiateur.

Compression XML et transfert brut Transférer des documents XML entre les adaptateurs et les médiateurs semble être coûteux. Chaque XTuple est codé dans un message XML et envoyé sur le réseau. Le message XML est alors analysé au niveau client et transformé en interne en un descripteur de XTuple et en arbres XML au fil du flux d'évènements. Ainsi, le nombre de messages est important et la durée de la transformation est longue. Pour gagner en nombre de messages, nous pourrions employer le transfert en masse, et envoyer plusieurs messages dans un bloc. Le nombre de messages par bloc devrait être accordé de telle sorte que le pipeline sur le client puisse continuer à travailler sans à-coup. Néanmoins, ceci n'empêche pas l'analyse et la transformation des messages très longs. C'est de toute façon inhérent à XML et ceci dégradera toujours les exécutions. Une solution est d'employer un format compressé pour transférer les XTuples. Les schémas de XTuples sont connus par le client et le serveur sous la forme d'une liste de chemins. Les types de valeurs (feuilles des arbres XML) sont également connus par des schémas XML. Ainsi, un mécanisme évident de compression consiste à envoyer un XTuple comme une séquence d'identifiants de chemin (16 bits sont suffisants) suivie de la valeur de feuille codée selon son type. L'analyse sera alors une tâche évidente. Cependant, nous nous éloignons alors de la philosophie de XML et de la généralité du mécanisme de communication. Bien que ce soit un peu contraire aux principes de XML, nous croyons qu'un dispositif de compression permettant d'économiser du temps d'analyse est crucial.

Cache sémantique et indexation des données Un autre axe de recherche consisterait à nous intéresser à l'utilisation du cache semi-structuré local en nous interrogeant sur le placement optimal des données suivant la façon dont elles seront interrogées. Des vues matérialisées sur le cache semi-structuré local serait envisageable.

On pourrait aussi étudier l'utilisation des index du cache de l'entrepôt natif pour améliorer les recherches, notamment les recherches textuelles.

Modèle de coût Les formules de coût des XOpérateurs devront être améliorées. Elles devront aussi prendre en compte la gestion du cache et la gestion des capacités des sources.

9.3.3 Extension

En plus de ces travaux portant sur l'optimisation, plusieurs d'extensions sont à l'étude et devraient être approfondies.

Fonctions externes Pour pouvoir intégrer des sources multimédia, des fonctions externes pourraient être pris en considération.

Mises à jour La gestion des mises à jour est un point important. Pour cela, dans les propositions de XQuery un certain nombre d'instructions sont proposées. Il faudrait pouvoir les utiliser et se servir de telles requêtes de mises à jour pour réactualiser le cache du médiateur. Un mécanisme de déclencheur (*trigger*) par temporisation ou par évènement pourrait alors y être associé.

Utilisation sur du web sémantique Des adaptateurs [Kou 2003] pour des recherches « plein-texte » et des recherches sémantiques sont à l'étude, en particulier pour la conception d'un adaptateur de page web.

Bibliographie

[Abiteboul *et al.* 1997] S. Abiteboul, D. Quass, J. McHugh, J. Widom, et J. Wiener. The Lorel Query Language for Semi-Structured Data. *Journal of the Digital Library*, 1 (1) :68–88, april 1997.

[Abiteboul *et al.* 1998] S. Abiteboul, J. Widom, et T. Lahiri. An Unified Approach for Querying Structured Data and XML, 1998. http ://www.w3.org/TandS/QL/SL98/pp/serge.html.

[Abiteboul 1997] S. Abiteboul. Querying Semistructured Data. In *Proceeding of the 6th International Conference on Database Theory*, Delphi, Greece, 1997.

[Abiteboul 1998] S. Abiteboul. Semistructured Data Tutorial, 1998.

[Aboulnaga *et al.* 2001] A. Aboulnaga, A.R. Alameldeen, et J.F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *The VLDB Journal*, pages 591–600, 2001.

[Adali *et al.* 1996] S. Adali, K. Candan, et Y. Papakonstantinou. Query Caching and Optimization in Distributed Mediator Systems. In *ACM SIGMOD International Conference on Management of Data*, pages 137–148, Montreal, Canada, 1996.

[Ahmed *et al.* 1987] R. Ahmed, J. Albert, W. Du, W. Kent, W. Litwin, et M. Shan. An Overview of Pegasus. In *IEEE Conference on Data Engineering*, Vienna, April 1987.

[Ausbrooks *et al.* 2002] R. Ausbrooks, S. Buswell, D. Carlisle, S. Dalmas, S. Devitt, A. Diaz, R. Hunter, P. Ion, R. Miner, N. Poppelier, B. Smith, N. Soiffer, R. Sutor, et S. Watt. Mathematical Markup Language (MathML) Version 2.0, 2002. http ://www.w3.org/TR/MathML2/.

[BEA 2002] BEA. BEA Liquid Data for WebLogic - Product Overview, octobre 2002.

[Beech *et al.* 1999] D. Beech, A. Malhotra, et M. Rys. A Formal Data Model and Algebra for XML. Septembre 1999.

[Biron et Malhotra 2000] P. Biron et A. Malhotra. XML Schema Part 2 : Datatypes, October 2000.

[Böhme et Rahm 2001] T. Böhme et E. Rahm. In *Proceedings of German Database Conference (BTW2001)*, Oldenburg, Mars 2001. Springer.

[Bonifati et Ceri 2000] A. Bonifati et S. Ceri. Comparative Analysis of Five XML Query Languages. *SIGMOD Record*, 29(1) :68–79, 2000.

[Bornhovd 1998] C. Bornhovd. *MIX - A Representation Model for the Integration of Web-based Data.* Technical report, Dep. CS, Darmstadt University of Technology, Germany, 1998.

[Bosak *et al.* 1998] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, C. Sperberg-McQueen, L. Wood, et J. Clark. W3C XML Specification DTD, 1998.

[Bouganim *et al.* 1999] L. Bouganim, T. Chan-Sine-Ying, T.-T. Dang-Ngoc, J.-L. Darroux, G. Gardarin, et F. Sha. MIROWeb : Integrating Multiple Data Sources through Semistructured Data Types. In *25th International Conference on Very Large Data Bases*, pages 750–753, Edinburgh, Scotland, 1999.

[Bourret 1999] R. Bourret. XML And Databases, September 1999.

[Bourret 2000] R. Bourret. XML Database Products, March 2000.

[Bray *et al.* 1998] T. Bray, J. Paoli, et C. Sperberg-MacQueen. Extensible Markup Language (XML) 1.0 (W3C Recommendation), 1998.

[Caravel 1998] Caravel. LeSelect, 1998.

[Carey *et al.* 1993] M.J. Carey, D.J. DeWitt, et J.F. Naughton. The OO7 Benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2) :12–21, 1993.

[Carey *et al.* 2000a] M.J. Carey, D. Florescu, Z.G. Ives, Y. Lu, J. Shanmugasundaram, E.J. Shekita, et S.N. Subramanian. XPERANTO : Publishing Object-Relational Data as XML. In *WebDB (Informal Proceedings)*, 2000.

[Carey *et al.* 2000b] M.J. Carey, J. Kiernan, J. Shanmugasundaram, E.J. Shekita, et S.N. Subramanian. XPERANTO : Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[Carey 1995] M. Carey. Towards Heterogeneous Multimedia Information Systems : The Garlic Approach. In *5th Workshop on Research Issues in Data Engineering - Distributed Object Management (RIDE-DOM)*, pages 124–131, Taipei, Taiwan, 1995.

[Chandra et Merlin 1977] A.K. Chandra et P.M. Merlin. Optimal Implementation of Conjonctive Queries in Relational Databases. In *Proceeding of the Ninth Annual ACM Symposium on the Theory of Computing*, pages 77–90, 1977.

[Chawathe *et al.* 1994] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, et J. Widom. The TSIMMIS Project - Integration of Heterogeneous Information Sources. In *10th Anniversary Meeting of Information Processing Society of Japan*, Tokyo, Japan, 1994.

[Chidlovskii *et al.* 1999] B. Chidlovskii, C. Roncancio, et M.-L. Schneider. Optimizing Web Queries through Semantic Caching. In *Proceeding 15emes Journees Bases de Donnees Avancees, BDA*, pages 23–40, 1999.

[Christophides *et al.* 1994] V. Christophides, S. Abiteboul, S. Cluet, et M. Scholl. From structured documents to novel query facilities. In *In Proc. of ACM SIGMOD Conf. on Management of Data*, pages 313–324, Minneapolis, Minnesota, Mai 1994.

[Christophides *et al.* 2000] V. Christophides, S. Cluet, et J. Simeon. On Wrapping Language and efficient XML Integration. In *ACM SIGMOD Conference on Management of Data*, 2000.

[Clark et Deach 2001] J. Clark et S. Deach. Extensible Stylesheet Language (XSL), 2001.

[Clark et DeRose 1999] J. Clark et S. DeRose. XML Path Language (XPath) Version 1.0, 1999.

[Cluet 1998] S. Cluet. Your Mediators need Data Conversion ! In *International Conference on Management of Data ACM SIGMOD*, Seattle, 1998.

[Codd 1972] E.F. Codd. *Relational Completness of Databases Sublanguages.* Prentice Hall, Englewood, Cliffs, NJ, 1972.

[Consortium 2000] World-Wide-Web Consortium. The XML Query Algebra, Decembre 2000.

[Cutting et Pedersen 1990] D. Cutting et J. Pedersen. Optimizations for Dynamic Inverted Index Maintenance. In *Proceedings of the 13th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, 1990.

[Dar *et al.* 1996] S. Dar, M.J. Franklin, B. Jonsson, D. Srivastava, et M. Tan. Semantic Data Caching and Replacement. In *Proceedings of the 22nd Conference on Very Large Databases (VLDB)*, pages 330–341, 1996.

[D.Draper *et al.* 2001] D.Draper, A.Y. Halevy, et D.S. Weld. The Nimble XML Data Integration System. In *ICDE*, pages 155–160, 2001.

[Deutsch *et al.* 1998] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, et D. Suciu. XML-QL : A Query Language for XML, 1998.

[Draper *et al.* 2001] D. Draper, A.Y. Halevy, et D.S. Weld. The Nimble Integration Engine. In *SIGMOD Conference*, 2001.

[Du *et al.* 1992] W. Du, R. Krishnamurthy, et M.-C. Shan. Query optimization in a heterogeneous DBMS. In *Proceeding of the 18th International Conference on Very Large Data Bases (VLDB)*, pages 277–291, Vancouver, Canada, 1992.

[Fankhauser *et al.* 1998] P. Fankhauser, G. Gardarin, M. Lopez, J. Muntz, et A. Tomasic. Experiences in Federated Databases : From IRO-DB to MIRO-Web. In *Proceedings of the 24th International Conference on Very Large Data Bases*, New-York, USA, August 1998.

[Fegaras et Maier 1995] L. Fegaras et D. Maier. An Algebraic Framework for physical OODB Design. In *proceeding of the 5th International Workshop on Database Programming Languages*, Italie, 1995.

[Fernandez *et al.* 1998] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, et D. Suciu. Catching the Boat with Strudel : Experiences with a Web-Site Management System. In *SIGMOD Conference*, pages 414–425, 1998.

[Fernandez *et al.* 2001] M. Fernandez, J. Simeon, et P. Walder. A Semi-Monad for Semi-structured Data. In *International Conference on Database Theory*, Janvier 2001.

[Fernández et Siméon 2000] M. Fernández et J. Siméon. A Data Model and Algebra for XML Query, 2000.

[Florescu *et al.* 1999] D. Florescu, A. Levy, I. Manolescu, et D. Suciu. Query Optimization in the Presence of Limited Access Patterns. In *In Proceeding of ACM SIGMOD Conf. on Management of Data*, pages 311–322, 1999.

[Florescu *et al.* 2000] D. Florescu, I. Manolescu, D. Kossmann, et F. Xhumari. Agora : Living with XML and Relational. In *Proceeding of the Conference on Very Large Data Base*, Cairo, Egypt, Février 2000.

[Franklin *et al.* 1993] M. Franklin, M. Carey., et M. Livny. Local Disk Caching for Client-Server Database Systems. In *Proceedings of the 19th Conference on Very Large Databases (VLDB)*, Dublin, 1993.

[Galanis *et al.* 2001] L. Galanis, E. Viglas, D.J. DeWitt, J.F. Naughton, et D. Maier. *Following the paths of XML Data : an Algebraic Framework for XML Query Evaluation.* Technical report, 2001.

[Garcia-Molina *et al.* 1994] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, et J. Widom. Integrating and Accessing Heterogeneous Information in TSIMMIS. In *10th Anniv. Meeting of Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, 1994.

[Gardarin *et al.* 1994] G. Gardarin, B. Finance, P. Fanhauser, et W. Klas. IRO-DB : a Distributed System Federating Object and Relational Databases. *Object Oriented Multibase Systems : A Solution for Advanced Applications*, 1994.

[Gardarin *et al.* 1996a] G. Gardarin, J.-R. Gruser, et Z.-H. Tang. Cost-based Selection of Path Expression Processing Algorithms in Object-Oriented Databases. In *The VLDB Journal*, pages 390–401, 1996.

[Gardarin *et al.* 1996b] G. Gardarin, F. Sha, et Z.-H. Tang. Calibrating the Query Optimizer Cost Model of IRO-DB. In *Proceeding of the 22sd International Conference on Very Large Data Bases (VLDB)*, pages 378–389, Mumbai (Bombay), Inde, 1996.

[Gardarin *et al.* 1999] G. Gardarin, F. Sha, et T.-T. Dang-Ngoc. XML-Based Components for Federating Multiple Heterogeneous Data Sources. In *18th International Conference on Conceptual Modeling*, volume 1728 of *Lecture Notes in Computer Science*, pages 506–519, Paris, France, 1999. Springer.

[Gardarin et Yoon 1996] G. Gardarin et S. Yoon. Hyoql : A query language for structured hypermedia documents. *The International Journal of Microcomputer Applications*, 1996.

[Gardarin 1997] G. Gardarin. Multimedia Federated Databases on Intranets : Web-Enabling IRO-DB. In *8th International Conference on Database and Expert Systems Applications (DEXA'97)*, France, 1997.

[Gardarin 2002] G. Gardarin. *XML.* Dunod, 2002.

[Goldfarb 1991] C. Goldfarb. *The SGML Handbook.* Clarendon Press, 1991.

[Goldman *et al.* 1999] R. Goldman, J. McHugh, et J. Widom. From Semistructured Data to XML : migrating the Lore Data Model and Query Language. In *Proceeding of the 2nd International Workshop on the Web and Databases (WebDB'99)*, Philadelphia, Pennsylvania – USA, June 1999.

[Goldman et Widom 1997] R. Goldman et J. Widom. DataGuides : Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceeding of the Conference on Very Large Data Base (VLDB)*, Athens, Greece, 1997.

[Grahne et Lakshmanan 2000] G. Grahne et L.V. S. Lakshmanan. On the Difference between Navigating Semi-structured Data and Querying It. *Lecture Notes in Computer Science*, 1949 :271–??, 2000.

[Haas *et al.* 1997] L.M. Haas, D. Kossmann, E.L. Wimmers, et J. Yang. Optimizing Queries Across Diverse DataSources. In *23th International Conference on Very Large Data Bases*, pages 276–285, Athens, Greece, 1997.

[Halevy 2000] A. Halevy. Logic-based techniques in Data Integration. *Logic Based Artificial Intelligence*, 2000.

[Huck *et al.* 1999] G. Huck, I. Macherius, et P. Fankhauser. PDOM : Lightweight Persistency Support for the Document Object Model. In *In proceeding of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, Denver, 1999.

[IBM 2002] IBM. XTables : Bridging Relation Technology and XML, 2002.

[Inc. 1997] Sun Microsystem Inc. JDBC : A Java SQL API, 1997.

[Jagadish *et al.* 2001] H.V. Jagadish, L.V.S.Lakshmanan, D.Srivastava, et K.Thompson. TAX : A Tree Algebra for XML. In *Proceeding DBPL Conference*, Rome, Italy, Septembre 2001.

[Kanne et Moerkotte 2000] C.-C. Kanne et G. Moerkotte. Efficient Storage of XML Data. In *ICDE*, page 198, 2000.

[Keller et Basu 1996] A.M. Keller et J. Basu. A Predicate-based Caching Scheme for Client-Server Database architectures. *VLDB Journal : Very Large Data Bases*, 5(1) :35–47, 1996.

[Kirk *et al.* 1995] T. Kirk, A. Levy, et D. Srivastava. *The Information Manifold.* Technical report, AT&T Bell Laboratories, 1995.

[Kou 2003] H. Kou. *Génération d'Adaptateurs Web intelligents à l'Aide de Techniques de Fouille de Texte.* PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, juillet 2003.

[Landers et Rosenberg 1982] T. Landers et R.L. Rosenberg. An Overview of MULTIBASE. *Distributed Databases*, 1982.

[Levy *et al.* 1996] A. Levy, A. Rajaraman, et J. Ordille. Querying Heterogeneous Information Sources Using Source Description. In *Proceeding of International Conference on Very Large Data Base*, Bombay, 1996.

[Li *et al.* 1998] C. Li, R. Yerneni, V. VassalosV., H. Garcia-Molina, Y. Papakonstantinou, J.D. Ullman, et M. Valiveti. Capability Based Mediation in TSIMMIS. In *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 564–566, Seattle, Washington, 1998.

[Luascher *et al.* 1999] B. Luascher, Y. Papakonstantinou, P. Velikhov, et V. Vianu. View definition and DTD inference for XML, Janvier 1999.

[Luo *et al.* 2001] Q. Luo, J.F. Naughton, R. Krishnamurthy, P. Cao, et Y. Li. Active Query Caching for Database Web Servers. *Lecture Notes in Computer Science*, 1997 :92–??, 2001.

[Maitre *et al.* 1997] J. Le Maitre, E. Murisasco, et M. Rolbert. From annotated corpora to databases : The SgmlQL language. In John Nerbonne, editor, *Linguistic Databases*, pages 37–58. CSLI Publications, Stanford, California, 1997.

[Manolescu *et al.* 2001] I. Manolescu, D. Florescu, et D. Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pages 241–250, Rome, Italie, Septembre 2001.

[McHugh *et al.* 1997] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, et J. Widom. LORE : A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3) :54–66, 1997.

[McHugh et Widom 1999a] J. McHugh et J. Widom. *Query Optimization for SemiStructured Data.* Technical report, Standford University Database Group, Février 1999.

[McHugh et Widom 1999b] J. McHugh et J. Widom. Query Optimization for XML. In *Proceeding of the 25th International Conference on Very Large Databases*, Edinburgh, Ecosse, Septembre 1999.

[Milo et Suciu 1999] T. Milo et D. Suciu. Index Structures for Path Expressions. In *Intl Conf. on Database Theory*, 1999.

[Mullen et Elmagarmid 1993] J.G. Mullen et A. Elmagarmid. InterSQL : a Multidatabase Transaction Programming Language. In *Workshop on Database Programming Language*, 1993.

[Naacke *et al.* 1998] H. Naacke, G. Gardarin, et A. Tomasic. Leveraging Mediator Cost Models with Heterogeneous Data Sources. In *ICDE*, 1998.

[Naacke 1999] H. Naacke. *Modèles de Coût pour Médiateurs de Bases de Données Hétérogènes.* PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, septembre 1999.

[Nambiar *et al.* 2001] U. Nambiar, Z. Lacroix, S. Bressan, M. Lee, et Y. Li. *Benchmarking XML Management Systems : The XOO7 Way.* Technical report, Dept of Computer Science, Arizona State University., 2001.

[Nestorov *et al.* 1997] S. Nestorov, S. Abiteboul, et R. Matwani. Inferring Structure in Semistructured Data. In *Proceeding of the Workshop on Management of Semistructured Data*, Arizona, USA, May 1997.

[Nimble 2002a] Nimble. Next Generation Data Integration, 2002.

[Nimble 2002b] Nimble. Nimble Datasheet, 2002.

[Nimble 2002c] Nimble. Nimble Integration Suite, 2002.

[Papakonstantinou *et al.* 1995] Y. Papakonstantinou, H. Garcia-Molina, et J. Widom. Object Exchange across Heterogeneous Information Sources. In *Int. Conf. on Data Engineering*, Taipei, 1995.

[Paparizos *et al.* 2002] S. Paparizos, S. Al-Khalifa, H.V. Jagadish, A. Nierman, et Y. Wu. *A Physical Algebra for XML.* Technical report, 2002.

[Q.Zhu et Larson 1998] Q.Zhu et P.-A Larson. Solving Local Cost Estimation Problem for Global Query Optimization in Multidatabase Systems. *Distributed and Parallel Databases*, 6 :373–421, 1998.

[Robie *et al.* 1998] J. Robie, J. Lapp, et D. Schach. XML Query Language (XQL), 1998.

[Robie *et al.* 2000] J. Robie, D. Chamberlin, et D. Florescu. Quilt : an XML Query Langage, March 2000.

[Roth *et al.* 1999] M.-T. Roth, F. Ozcan, et L.M. Haas. Cost Models DO Matter : Providing Cost Information for diverse Data Sources in a Federated System. In *25th International Conference on Very Large DataBases*, pages 599–610, Edinburgh, Scotland, 1999.

[Schmidt *et al.* 2001] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, et R. Busse. *The XML Benchmark Project.* Technical Report INS-R0103, CWI, April 2001.

[Schning et Wsch 2000] H. Schning et J. Wsch. Tamino – An Internet Database System. In *Proceeding of the 7th International Conference on Extending Database Technology (EDBT*, Konstanz, March 2000.

[Sha *et al.* 1999] F. Sha, G. Gardarin, et L. Némirovski. Managing Semistructured Data in Object-Relational Database. In *Proceeding, 15ème journée Bases de Données avancées*, 1999.

[Shoens *et al.* 1993] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, et J. Thomas. The Rufus System : Information Organization for Semi-Structured Data. In R. Agrawal, S. Baker, et D. Bell, editors, *19th International Conference on Very Large Data Bases*, pages 97–107, Dublin, Ireland, August 1993.

[Subrahmanian *et al.* 1997] V. Subrahmanian, S. Adali, A. Brink, R. Emery, J. Lu, A. Rajput, T. Rogers, R. Ross, et C. Ward. HERMES : Heterogeneous Reasoning and Mediator System, 1997.

[Suciu 1998] D. Suciu. Semistructured Data and XML. In *International Conference on Foundations of Data Organization*, 1998.

[Templeton *et al.* 1987] M. Templeton, D. Brill, S.K. Dao, E. Lund, P. Ward, A.L. Chen, et R. MacGregor. Mermaid - a Front-End to Distributed Heterogeneous Databases. In *IEEE Conference on Data Engineering*, Vienna, April 1987.

[Thompson *et al.* 2001] H. Thompson, D. Beech, M. Maloney, et N. Mendelsohn. XML Schema Part 1 : Structures, May 2001.

[Tomasic *et al.* 1996] A. Tomasic, L. Raschid, et P. Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. In *International conference on Distributed Computing Systems*, Hong Kong, 1996.

[Tomasic *et al.* 1997] A. Tomasic, R. Amouroux, P. Bonnet, O. Kapitskaia, H. Naacke, et L. Raschid. The Distributed Information Search Component (DISCO) and the World Wide Web. In Joan Peckham, editor, *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 546–548, Tucson, Arizona, May 1997. ACM Press.

[TPC] TPC. TPC : Transaction Processing Performance Council.

[Ullman 1989] J.D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1989.

[Vassalos et Papakonstantinou 1997] V. Vassalos et Y. Papakonstantinou. Describing and Using Capabilities of Heterogeneous Sources. In *Proceeding of International Conference on Very Large Data Base*, Athens, Greece, August 1997.

[W3C 2001] W3C. An XML Query Language (XQuery 1.0), 2001.

[Wiederhold 1992] G. Wiederhold. Mediators in the Architecture of Future Information System. *Computer*, 25 (3) :38–49, 1992.

[Wu et Jagadish 2002] Y. Wu et H.V. Jagadish. *Query Optimization for XML.* Technical report, 2002.

[Yao 1977] S.B. Yao. Approximating the Number of Accesses in Database Organization). 20 :260, 1977.

[Yeh et Dang-Ngoc 2001] T.-S. Yeh et T.-T. Dang-Ngoc. *ReposiX : Structure de données pour le stockage.* Technical report, PRiSM, 2001.

[Zaniolo 1985] C. Zaniolo. The Representation and Deductive Retrieval of Complex Objects. In *Proceedings of 11International Conference on Very Large Data Bases (VLDB)*, Stockholm, 1985.

[Zhu *et al.* 2000a] Q. Zhu, S. Motheramgari, et Y. Sun. Cost estimation for large queries via fractional analysis and probabilistic approach in dynamic multidatabase environments. In *Proceeding of DEXA*, volume 1873, 2000.

[Zhu *et al.* 2000b] Q. Zhu, Y. Sun, et S. Motheramgari. Developing Cost Models with Qualitative Variables for Dynamic Multidatabase Environment. In *Proceeding of the 16th ICDE*, pages 413–424, 2000.

[Zhu et Larson 1996] Q. Zhu et P.-A Larson. Building Regression Cost Models for Multidatabase systems. In *Porceeding of the 4th International Conference on Parallell and Distributed Information Systems*, pages 220–231, 1996.

[Zhu 1995] Q. Zhu. *Estimating Local Cost Parameters for Global Query Optimization in a Multidatabase System.* PhD thesis, University of Waterloo, 1995.

Annexe A

Coût

```
<xs:schema targetNamespace="http://pegase.prism.uvsq.fr/cout"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:math="http://www.w3.org/1998/Math/MathML">

<xs:element name="costmodel" minOccurs="1" maxOccurs="1">
 <xs:complexType>
  <xs:sequence>

   <xs:element name="statistics" minOccurs="0" maxOccurs="1">
    <xs:complexType>
     <xs:sequence>

      <xs:element name="system" minOccurs="0" maxOccurs="1">
       <xs:complexType>
        <xs:sequence>

         <math:element name="math" type=""
                       minOccurs="0" maxOccurs="unbounded"/>

        </xs:sequence>
       </xs:complexType>
      </xs:element>

      <xs:element name="collection"
                  minOccurs="0" maxOccurs="1">
       <xs:complexType>
        <xs:sequence>

         <math:element name="math" type=""
                       minOccurs="0" maxOccurs="unbounded"/>

        </xs:sequence>
       </xs:complexType>
      </xs:element>
```

```
      <xs:element name="user-defined" minOccurs="0" maxOccurs="1">
       <xs:complexType>
        <xs:sequence>

         <math:element name="math" type="" minOccurs="0"
                       maxOccurs="unbounded"/>

        </xs:sequence>
       </xs:complexType>
      </xs:element>

     </xs:sequence>
    </xs:complexType>
   </xs:element>

   <xs:element name="formulas" minOccurs="0" maxOccurs="1">
    <xs:complexType>
     <xs:sequence>

      <xs:element name="user-defined" minOccurs="0" maxOccurs="1">
       <xs:complexType>
        <xs:sequence>

         <math:element name="math" type=""
                       minOccurs="0" maxOccurs="unbounded"/>

        </xs:sequence>
       </xs:complexType>
      </xs:element>

      <xs:element name="generic" minOccurs="0" maxOccurs="1">
       <xs:complexType>
        <xs:sequence>

         <math:element name="math" type=""
                       minOccurs="0" maxOccurs="unbounded"/>

        </xs:sequence>
       </xs:complexType>
      </xs:element>

      <xs:element name="operators"
                  minOccurs="0" maxOccurs="1">
       <xs:complexType>
        <xs:sequence>

         <math:element name="math" type=""
                       minOccurs="0" maxOccurs="unbounded"/>

        </xs:sequence>
       </xs:complexType>
      </xs:element>

     </xs:sequence>
    </xs:complexType>
   </xs:element>


</xs:schema>
```

Annexe B

Capacité

```
<xs:schema targetNamespace="http://pegase.prism.uvsq.fr/capacite"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:element name="ruleset" minOccurs="0" maxOccurs="1">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="rule" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>

       <xs:element name="permission" type="xs:string"
                   minOccurs="1" maxOccurs="1"/>

       <xs:element name="relationalop" type="xs:string"
                   minOccurs="0" maxOccurs="1"/>

       <xs:element name="collection1" type="xs:string"
                   minOccurs="0" maxOccurs="unbounded"/>

       <xs:element name="attribute1" type="xs:string"
                   minOccurs="0" maxOccurs="unbounded"/>

       <xs:element name="operator" type="xs:string"
                   minOccurs="0" maxOccurs="unbounded"/>

       <xs:element name="collection2" type="xs:string"
                   minOccurs="0" maxOccurs="unbounded"/>

       <xs:element name="attribute2" type="xs:string"
                   minOccurs="0" maxOccurs="unbounded"/>

      </xs:sequence>
      <xs:attribute name="num" type="xs:int"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

</xs:schema>
```

Annexe C

Structure des tables TPCR

Cette annexe présente les différentes tables définies par le benchmark TPC-R. Certaines sont transformées suivant un schéma XML pour les besoins de nos tests.

C.1 Table PARTSUPP

Nom de colonne	Type de données requis	Commentaire
PS_PARTKEY	identifiant	Référence clef étrangère P_PARTKEY
PS_SUPPKEY	identifiant	Référence clef étrangère S_SUPPKEY
PS_AVAILQTY	entier	
PS_SUPPLYCOST	décimal	
PS_COMMENT	texte de taille variable 199	

Clef primaire composée : PS_PARTKEY, PS_SUPPKEY

C.2 Table CUSTOMER

Nom de colonne	Type de données requis	Commentaire
C_CUSTKEY	identifiant	SF*150,000 sont générés
C_NAME	texte de taille variable 25	
C_ADDRESS	texte de taille variable 40	
C_NATIONKEY	identifiant	Référence clef étrangère N_NATIONKEY
C_PHONE	texte de taille fixe 15	
C_ACCTBAL	décimal	
C_MKTSEGMENT	texte de taille fixe 10	
C_COMMENT	texte de taille variable 117	

Clef primaire : C_CUSTKEY

C.3 Table LINEITEM

Nom de colonne	Type de données requis	Commentaire
L_ORDERKEY	identifiant	Référence clef étrangère O_ORDERKEY
L_PARTKEY	identifiant	Référence clef étrangère P_PARTKEY, référence clef étrangère composée (PS_PARTKEY, PS_SUPPKEY) avec L_SUPPKEY
L_SUPPKEY	identifiant	Référence clef étrangère S_SUPPKEY, référence clef étrangère composée (PS_PARTKEY, PS_SUPPKEY) avec L_PARTKEY
L_LINENUMBER	entier	
L_QUANTITY	décimal	
L_EXTENDEDPRICE	décimal	
L_DISCOUNT	décimal	
L_TAX	décimal	
L_RETURNFLAG	texte de taille fixe 1	
L_LINESTATUS	texte de taille fixe 1	
L_SHIPDATE	date	
L_COMMITDATE	date	
L_RECEIPTDATE	date	
L_SHIPINSTRUCT	texte de taille fixe 25	
L_SHIPMODE	texte de taille fixe 10	
L_COMMENT	variable text size 44	

Clef primaire composée : L_ORDERKEY, L_LINENUMBER

C.4 Table ORDERS

Nom de colonne	Type de données requis	Commentaire
O_ORDERKEY	identifiant	SF*1,500,000 are sparsely populated
O_CUSTKEY	identifiant	Référence clef étrangère C_CUSTKEY
O_ORDERSTATUS	texte de taille fixe 1	
O_TOTALPRICE	décimal	
O_ORDERDATE	date	
O_ORDERPRIORITY	texte de taille fixe 15	
O_CLERK	texte de taille fixe 15	
O_SHIPPRIORITY	entier	
O_COMMENT	texte de taille variable 79	

Clef primaire : O_ORDERKEY

C.5 Table SUPPLIER

Nom de colonne	Type de données requis	Commentaire
S_SUPPKEY	identifiant	SF*10,000 sont générés
S_NAME	texte de taille fixe 25	
S_ADDRESS	texte de taille variable 40	
S_NATIONKEY	identifiant	Référence clef étrangère N_NATIONKEY
S_PHONE	texte de taille fixe 15	
S_ACCTBAL	décimal	
S_COMMENT	texte de taille variable 101	

Clef primaire : S_SUPPKEY

Cette table est transformée en un document XML suivant le schéma :

```
<xs:schema targetNamespace="http://supplier.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="SUPPLIER">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="S_ID">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="S_SUPPKEY" type="xs:positiveInteger" />
              <xs:element name="S_NAME" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="S_CONTACT">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="S_LOCALISATION">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="S_ADDRESS" type="xs:string" />
                    <xs:element name="S_NATIONKEY" type="xs:positiveInteger" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="S_PHONE" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="S_DETAIL">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="S_ACCTBAL" type="xs:decimal" />
              <xs:element name="S_COMMENT" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

C.6 Table NATION

Nom de colonne	Type de données requis	Commentaire
N_NATIONKEY	identifiant 25 nations sont générés	
N_NAME	texte de taille fixe 25	
N_REGIONKEY	identifiant	Référence clef étrangère R_REGIONKEY
N_COMMENT	texte de taille variable 152	

Clef primaire : N_NATIONKEY

Cette table est transformée en un document XML suivant le schéma :

```
<xs:schema targetNamespace="http://region.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="REGION">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="R_ID">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="R_NAME" type="xs:string" />
              <xs:element name="R_REGIONKEY" type="xs:positiveInteger" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="R_COMMENT" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

C.7 Table REGION

Nom de colonne	Type de données requis	Commentaire
R_REGIONKEY	identifiant	5 regions sont générés
R_NAME	texte de taille fixe 25	
R_COMMENT	texte de taille variable 152	

Clef primaire : R_REGIONKEY

Cette table est transformée en un document XML suivant le schéma :

```
<xs:schema targetNamespace="http://nation.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="NATION">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="N_NAME" type="xs:string" />
        <xs:element name="N_KEYS">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="N_NATIONKEY" type="xs:positiveInteger" />
              <xs:element name="N_REGIONKEY" type="xs:positiveInteger" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

C.8 Table PART

Nom de colonne	Type de données requis	Commentaire
P_PARTKEY	identifiant	SF*200,000 sont générés
P_NAME	texte de taille variable 55	
P_MFGR	texte de taille fixe 25	
P_BRAND	texte de taille fixe 10	
P_TYPE	texte de taille variable 25	
P_SIZE	entier	
P_CONTAINER	texte de taille fixe 10	
P_RETAILPRICE	décimal	
P_COMMENT	texte de taille variable 23	

Clef primaire : P_PARTKEY

Cette table est transformée en un document XML suivant le schéma :

```
<xs:schema targetNamespace="http://part.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="PART">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="P_IDENTIFICATION">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="P_PARTKEY" type="xs:positiveInteger" />
              <xs:element name="P_NAME" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="P_DESIGNATION">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="P_MFGR" type="xs:string" />
              <xs:element name="P_OBJECT">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="P_SIZE" type="xs:positiveInteger" />
                    <xs:element name="P_CONTAINER" type="xs:string" />
                    <xs:element name="P_TYPE" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>

              <xs:element name="P_BRAND" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>

        <xs:element name="P_DETAIL">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="P_RETAILPRICE" type="xs:decimal" />
              <xs:element name="P_COMMENT" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Annexe D

Code d'exportation des adaptateurs de la validation

D.1 Adaptateur A1

primitive Card ("PARTSUPP") := 80000

primitive Card ("CUSTOMER") := 15000

```
<ds:datasource name="WrapperSQL1"
xmlns:ds="http://www.e-xmlmedia.com/2002/XMLizer/Datasource">

    <description>description</description>
    <url>jdbc:oracle:thin:@pikachu:1521:orcl</url>
    <user>DNTT1</user>
    <password>DNTT1</password>
    <catalog>
        <schema name="DNTT1" targetNamespace="http://DNTT1">
            <includes>
                <table name="PARTSUPP"/>
                <table name="CUSTOMER"/>
           </includes>
        </schema>
    </catalog>
</ds:datasource>
```

D.2 Adaptateur A2

primitive Card ("LINEITEM") := 600000

```
<ds:datasource name="WrapperSQL2"
xmlns:ds="http://www.e-xmlmedia.com/2002/XMLizer/Datasource">

    <description>description</description>
    <url>jdbc:oracle:thin:@pikachu:1521:orcl</url>
    <user>DNTT2</user>
    <password>DNTT2</password>
    <catalog>
        <schema name="DNTT2" targetNamespace="http://DNTT2">
            <includes>
                <table name="LINEITEM"/>
           </includes>
        </schema>
    </catalog>
</ds:datasource>
```

D.3 Adaptateur A3

primitive Card ("ORDERS") := 150000

```
<ds:datasource name="WrapperSQL3"
xmlns:ds="http://www.e-xmlmedia.com/2002/XMLizer/Datasource">

    <description>description</description>
    <url>jdbc:oracle:thin:@pikachu:1521:orcl</url>
    <user>DNTT3</user>
    <password>DNTT3</password>
    <catalog>
        <schema name="DNTT3" targetNamespace="http://DNTT3">
            <includes>
                <table name="ORDERS"/>
           </includes>
        </schema>
    </catalog>
</ds:datasource>
```

D.4 Adaptateur A4

primitive Card ("SUPPLIER") := 20000

D.5 Adaptateur A5

primitive Card ("PART") := 1000

D.6 Adaptateur A6

primitive Card ("NATION") := 25

primitive Card ("REGION") := 5

D.7 Mediateur M0

```
<accessor type="mediator" name="Mediator0">

<launcher type="jvm"/>

<specific>
</specific>

<subaccessors>
    <subaccessor name="WrapperSQL1">
       <driver>com.exmlmedia.extractor.ExtractorDriver</driver>
       <connection>
           xdbc:exml:extractor:file:/users/dntt/these/experience/conf/A1
       </connection>
    </subaccessor>

    <subaccessor name="WrapperSQL2">
       <driver>com.exmlmedia.extractor.ExtractorDriver</driver>
       <connection>
           xdbc:exml:extractor:file:/users/dntt/these/experience/conf/A2
       </connection>
    </subaccessor>

    <subaccessor name="WrapperSQL3">
       <driver>com.exmlmedia.extractor.ExtractorDriver</driver>
       <connection>
           xdbc:exml:extractor:file:/users/dntt/these/experience/conf/A3
       </connection>
    </subaccessor>

    <subaccessor name="WrapperSQLX1">
       <driver>com.exmlmedia.repository.RepositoryDriver</driver>
       <connection>
           xdbc:exml:repository:jdbc:oracle:thin:pikachu:1521:orcl</conn
       </connection>
       <login>DNTTX1</login>
       <passwd>DNTTX1</passwd>
    </subaccessor>

    <subaccessor name="WrapperSQLX2">
       <driver>com.exmlmedia.repository.RepositoryDriver</driver>
       <connection>
           xdbc:exml:repository:jdbc:oracle:thin:pikachu:1521:orcl</conn
       </connection>
       <login>DNTTX2</login>
       <passwd>DNTTX2</passwd>
    </subaccessor>

    <subaccessor name="WrapperSQLX3">
       <driver>com.exmlmedia.repository.RepositoryDriver</driver>
       <connection>
           xdbc:exml:repository:jdbc:oracle:thin:pikachu:1521:orcl</conn
       </connection>
       <login>DNTTX3</login>
       <passwd>DNTTX3</passwd>
    </subaccessor>
</subaccessors>

</accessor>
```

D.8 Mediateur M1

```
<accessor type="mediator" name="Mediator1">

<launcher type="jvm"/>

<specific>
</specific>

<subaccessors>
    <subaccessor name="Mediator2">
       <driver>com.exmlmedia.mediator.MediatorDriver</driver>
       <connection>
           xdbc:exml:mediator:file:/users/dntt/these/experience/conf/M2
       </connection>
    </subaccessor>
    <subaccessor name="Mediator3">
       <driver>com.exmlmedia.mediator.MediatorDriver</driver>
       <connection>
           xdbc:exml:mediator:file:/users/dntt/these/experience/conf/M3
       </connection>
    </subaccessor>
</subaccessors>

</accessor>
```

D.9 Mediateur M2

```
<accessor type="mediator" name="Mediator2">

<launcher type="jvm"/>

<specific>
</specific>

<subaccessors>
    <subaccessor name="WrapperSQL1">
       <driver>com.exmlmedia.extractor.ExtractorDriver</driver>
       <connection>
           xdbc:exml:extractor:file:/users/dntt/these/experience/conf/A1
       </connection>
    </subaccessor>

    <subaccessor name="WrapperSQL2">
       <driver>com.exmlmedia.extractor.ExtractorDriver</driver>
       <connection>
           xdbc:exml:extractor:file:/users/dntt/these/experience/conf/A2
       </connection>
    </subaccessor>

    <subaccessor name="WrapperSQL3">
       <driver>com.exmlmedia.extractor.ExtractorDriver</driver>
       <connection>
           xdbc:exml:extractor:file:/users/dntt/these/experience/conf/A3
       </connection>
    </subaccessor>
</subaccessors>

</accessor>
```

D.10 Mediateur M3

```
<accessor type="mediator" name="Mediator3">

<launcher type="jvm"/>

<specific>
</specific>

<subaccessors>
    <subaccessor name="WrapperSQLX1">
       <driver>com.exmlmedia.repository.RepositoryDriver</driver>
       <connection>
           xdbc:exml:repository:jdbc:oracle:thin:pikachu:1521:orcl
       </connection>
       <login>DNTTX1</login>
       <passwd>DNTTX1</passwd>
    </subaccessor>

    <subaccessor name="WrapperSQLX2">
       <driver>com.exmlmedia.repository.RepositoryDriver</driver>
       <connection>
           xdbc:exml:repository:jdbc:oracle:thin:pikachu:1521:orcl
       </connection>
       <login>DNTTX2</login>
       <passwd>DNTTX2</passwd>
    </subaccessor>

    <subaccessor name="WrapperSQLX3">
       <driver>com.exmlmedia.repository.RepositoryDriver</driver>
       <connection>
           xdbc:exml:repository:jdbc:oracle:thin:pikachu:1521:orcl
       </connection>
       <login>DNTTX3</login>
       <passwd>DNTTX3</passwd>
    </subaccessor>
</subaccessors>

</accessor>
```

On ne finit jamais un projet, on arrête juste de travailler dessus.

Résumé Contrairement aux données traditionnelles, les données semi-structurées sont irrégulières : des données peuvent manquer, des concepts similaires peuvent être représentés par différents types de données, et les structures même peuvent être mal connues. Cette absence de schéma prédéfini, permettant de tenir compte de toutes les données du monde extérieur, présente l'inconvénient de complexifier les algorithmes d'intégration des données de différentes sources.

Nous proposons une architecture de médiation basée entièrement sur XML. L'objectif de cette architecture de médiation est de fédérer des sources de données distribuées de différents types. Elle s'appuie sur le langage XQuery, un langage fonctionnel conçu pour formuler des requêtes sur des documents XML. Le médiateur analyse les requêtes exprimées en XQuery et répartit l'exécution de la requête sur les différentes sources avant de recomposer les résultats.

L'évaluation des requêtes doit se faire en exploitant au maximum les spécificités des données et permettre une optimisation efficace. Nous décrivons l'algèbre XAlgebre à base d'opérateurs conçus pour XML. Cette algèbre a pour but de construire des plans d'exécution pour l'évaluation de requêtes XQuery et traiter des tuples d'arbres XML.

Ces plans d'exécution doivent pouvoir être modélisés par un modèle de coût et celui de coût minimum sera sélectionné pour l'exécution. Dans cette thèse, nous définissons un modèle de coût pour les données semi-structurées adapté à notre algèbre.

Les sources de données (SGBD, serveurs Web, moteur de recherche) peuvent être très hétérogènes, elles peuvent avoir des capacités de traitement de données très différentes, mais aussi avoir des modèles de coût plus ou moins définis. Pour intégrer ces différentes informations dans l'architecture de médiation, nous devons déterminer comment communiquer ces informations entre le médiateur et les sources, et comment les intégrer. Pour cela, nous utilisons des langages basés sur XML comme XML-Schema et MathML pour exporter les informations de métadonnées, de formules de coûts et de capacité de sources. Ces informations exportées sont communiquées par l'intermédiaire d'une interface applicative nommée XML/DBC.

Enfin, des optimisations diverses spécifiques à l'architecture de médiation doivent être considérées. Nous introduisons pour cela un cache sémantique basé sur un prototype de SGBD stockant efficacement des données XML en natif.

Mots-clef : médiateur, adaptateur, modèle de coût, cache sémantique, données semi-structurées, XML, base de données hétérogènes, algèbre semi-structurées, optimisation de requêtes, MathML, XMLSchema, XML/DBC.

Abstract In contrast to the traditional data, semi-structured data are irregular: data may be missed, different data types may be for the similar concepts, and if any the structure may not be well-known. One lacks actually predefined schemas to describe the data of the real world. It makes it difficult to integrate the data from different sources.

We propose a mediator architecture entirely based on XML. The objective of the mediator architecture is to federate distributed and heterogeneous data sources. It relies on XQuery, the functional language that is designed to query across XML documents. The mediator parses the XQuery request, dispatch it to sources for evaluation and recompose results with additional query evaluation.

Query evaluation must be done by making best use of data specificity to carry out an efficient optimization. We present the algebra XAlgebra based on the operators designed for XML. This algebra aims to construct execution plans for the evaluation of XQuery and processes tuples of tree structure.

These execution plans must be evaluated by a cost model and one of them with the minimal cost will be selected. In this thesis, we define a cost model for semi-structured data that is designed for our algebra.

Since the data sources (DBMS, Web server, search engine, etc.) may be very heterogeneous, they can have different capabilities of processing data, and their cost models may also be defined with different precision. So, in order to integrate such information in the mediation architecture, we have to know how to communicate the information between the mediator and the sources and to integrate them. To do this, we use XML-based languages such as XML-schema and MathML to export the metadata, cost formula and the definitions of source capabilities. The exported information is transferred by an application interface called XML/DBC.

Finally, diverse optimizations specific to this mediator architecture must be considered. For this, we introduce a semantic cache based on the DBMS prototype that store natively and efficiently XML data.

Keywords : mediator, wrapper, cost-model, semantic caching, semi-structured data, XML, heterogeneous database, semi-structured algebra, query optimization, MathML, XMLSchema, XML/DBC.