UNIVERSITE PARIS SUD
SEPTEMBRE 2004

Vers un optimiseur générique des requêtes XQuery

Rapport de stage

Étudiant: Andrei ARION
DEA I3

Responsables de stage:

Véronique BENZAKEN
LRI
Université Paris XI

Ioana MANOLESCU
Équipe GEMO
INRIA-Futurs

Je tiens à remercier mes deux responsables Véronique Benzaken et Ioana Manolescu pour tout leurs investissements et soutien tout au long de ce stage. Egalement, je tiens a remercier à l'équipe BD de LRI, à l'équipe GEMO de INRIA et aussi aux toutes les personnes qui se sont investies dans le DEA pour que tout se passe pour le mieux, autant pour les cours, que pour les stages.

Table des matières

1 Introduction **4**
- 1.1 Contexte et motivation . . . 4
- 1.2 Objectif . . . 5
- 1.3 Plan . . . 5

2 XAM (XML Access Modules) **6**
- 2.1 La syntaxe des XAM . . . 6
  - 2.1.1 Spécification des nœuds dans les XAM . . . 7
  - 2.1.2 Spécification des arêtes dans les XAM . . . 7
  - 2.1.3 Exemples . . . 7
- 2.2 La sémantique des XAM . . . 9
  - 2.2.1 Préliminaires . . . 9
  - 2.2.2 Les informations fournies par un XAM dans l'absence des restrictions d'accès 10
    Exemple : La sémantique des XAM sans restrictions d'accès . . . 12
  - 2.2.3 Les informations fournies par un XAM dans la présence des restrictions d'accès 13
    Exemple : La sémantique des XAM dans la présence des restrictions d'accès . 13
- 2.3 Modéliser des stockages et des index en utilisant les XAM . . . 15
  - 2.3.1 Modéliser les stockages XML basses sur des systèmes relationnels . . . 16
  - 2.3.2 Modéliser des stockages XML natives . . . 18
  - 2.3.3 Modéliser des index par des XAM . . . 20
- 2.4 Restrictions sur les XAM . . . 23

3 Optimisation de requêtes XQuery sur les XAM **24**
- 3.1 Préliminaires . . . 24
  - 3.1.1 XQuery . . . 24
  - 3.1.2 Le fragment de XQuery utilisé . . . 25
- 3.2 Répondre a des requêtes XQuery en utilisant les XAM . . . 25
  - 3.2.1 Algorithme de re-écriture des requêtes XQuery en utilisant les XAM . . . 25
    Pas I. Trouver le sous-ensemble des XAM relevant pour chaque nœud de la requête . . . 26
    Pas II. Estimer le coût des alternatives de re-écriture et trier les alternatives dans l'ordre croissant des coûts. . . . 26
    Pas III. La construction des plans d'exécution . . . 26

4 Conclusions **30**
- 4.1 Perspectives . . . 30

Chapitre 1

Introduction

L'explosion récente d'Internet comme support de calculs globaux conduit à l'échange croissant d'informations. La nature hautement distribuée du réseau induit que les informations sont généralement collectées, traitées puis restructurées à partir de sources multiples et hétérogènes. Afin d'en permettre le traitement, ces informations sont publiées sous un même format. XML devient le standard de-facto pour l'échange et la manipulation de documents sur le Web. De nombreux facteurs plaident en faveur de son acceptation : le fait que les documents XML soient lisibles par un humain ainsi que leur nature auto-descriptive ; le processus actif de standardisation du W3C et le fait que de nombreuses entreprises, parmi les plus représentatives dans le domaine des technologies de l'information, l'aient d'ores et déjà adopté.

1.1 Contexte et motivation

Les résultats existants en matière d'optimisation de requêtes sur des données (XML) persistantes peuvent être classifiés en fonction du modèle sous-jacent de stockage et de traitement de requêtes. Des travaux se sont appuyés sur des systèmes de gestion de bases de données relationnelles pour stocker des documents XML, et ont traité des requêtes XML en les traduisant en SQL [5, 9, 11, 18]. En conséquence, tous les aspects reliés au placement des données, au stockage, et à l'optimisation de requêtes ont été délégués au système relationnel. Cette approche a ses limites, puisque des langages de requêtes XML plus récents, tels que XQuery, sont bien plus complexes, et très différents de SQL : par exemple, ces langages sont basés sur un modèle de données arborescent, pas sur des tuples, et favorisent des listes imbriquées comme brique de base de l'interrogation, et non pas des multi-ensembles plats, non-ordonnés, comme c'est le cas en SQL. Pour échapper à ces limitations, plusieurs systèmes de stockage persistant et de traitement de requêtes XML *natifs* ont été développés récemment [7, 10, 19]. Ces systèmes proposent des approches d'optimisation ciblées sur un sous-ensemble du langage de requêtes XQuery (ou autres) sur des stockages particuliers.

Toutes ces solutions ont en commun d'offrir un unique modèle de stockage. Or, ceci revêt un certain nombre de désavantages :

- Les utilisateurs qui ont adopté une telle solution (très spécifique) sont contraints de devoir toujours l'utiliser (même si elle ne correspond pas/plus aux besoins).
- Il n'y a pas de modèle de stockage ou indexation qui soit universellement le meilleur, car différentes applications (ou même différents jeux de documents XML) conduisent à des performances différentes dans chaque application. L'impossibilité d'adapter le stockage aux besoins de l'application conduit à un problème de performance.
- Enfin, des petites modifications sur le stockage et/ou le moteur d'exécution peuvent conduire

à devoir reécrire l'optimiseur de requêtes (car il est fortement lié aux structures du stockage).

1.2 Objectif

L'objectif de ce stage est d'étudier l'optimisation des requêtes indépendamment du stockage et des méthodes d'indexation proposées par tel ou tel système XML. Ce travail propose donc d'identifier des structures persistantes de stockage XML pertinentes pour répondre à des requêtes XQuery. Nous cherchons à explorer les problèmes spécifiques qui apparaissent et de dégager les limites en terme de performances que la généricité impose. Outre l'étude théorique, nous essayons d'évaluer notre approche par un prototype que nous avons commencé à implanter.

1.3 Plan

Dans un premier chapitre, nous allons présenter en détail les *Modules d'Accés XML (XML Access Modules ou XAM)*, en insistant sur la sémantique et sur la modélisation des modéles de stockage et d'index par nos XAM. Dans le chapitre suivant nous allons voir comment les XAM peuvent être utilisés pour repondre á des requêtes XQuery. Enfin nous allons présenter l'état d'avancement et les perspectives de ce travail.

Chapitre 2

XAM (XML Access Modules)

Pour concevoir un optimiseur indépendant du stockage, nous avons procédé en deux étapes :

1. Pour généraliser les différents modéles et techniques de stockage et indexations déjà proposés, nous avons différencié l'optimisation *physique* (choix de structures persistantes à utiliser, et des opérateurs physiques) de l'optimisation *logique* (choix des opérateurs logiques et leur ordonnancement).
2. L'optimisation physique s'appuie sur un ensemble de *descriptions de modules de stockage*, assez général pour exprimer une grande variété de modèles de stockage et de méthodes d'indexation.

Un descripteur de module de stockage "XML Access Module", ou XAM caractérise l'accès aux données qui peuvent être trouvées dans une structure de stockage de données XML (telle qu'une table relationnelle, un index organisé en arbre B+, un arbre XML persistent etc.) Du point de vue de l'optimisation, les modules de stockage et les indexes sont très similaires : ils fournissent l'accès aux données. Les XAM décrivent donc le contenu d'un stockage réel (sur disque ou en mémoire vive) indépendamment de l'implantation particulière de ce stockage.

2.1 La syntaxe des XAM

Chaque XAM correspond à une structure d'arbre. Nous modélisons donc un XAM par un triplet (NS,ES,o) qui décrit quelles sont les parties du document stockées dans le module de stockage. NS est l'ensemble des nœuds et ES est l'ensemble des arêtes du XAM, tandis qu'o est un marqueur qui indique si le stockage est ordonné (il reflete l'ordre des éléments du document initial) ou non. Par défaut nous considérons les stockages comme étant non-ordonnés.

Nous donnons la spécification des XAM par la grammaire suivante :

$$NS : -N* \tag{2.1}$$
$$N : -NE \mid NA \tag{2.2}$$
$$NE : -\ \mathsf{n}\ IDSpec?\ TSpec?\ VSpec?\ CSpec? \tag{2.3}$$
$$NA : -\ \mathsf{n}\ TSpec?\ VSpec?\ CSpec? \tag{2.4}$$
$$IDSpec : -\ (\mathsf{ID}\ (\mathsf{i} \mid \mathsf{o} \mid \mathsf{s} \mid \epsilon))?\ \ \mathsf{R}? \tag{2.5}$$
$$TSpec : -\ \mathsf{Tag}\ ?\ ([\mathsf{Tag}{=}c])?\ \mathsf{R}? \tag{2.6}$$
$$VSpec : -\ \mathsf{Val}\ ?\ ([\mathsf{Val}{=}c])?\ \mathsf{R}? \tag{2.7}$$
$$CSpec : -\ \mathsf{Cont} \tag{2.8}$$
$$ES : -E* \tag{2.9}$$
$$E : -\mathsf{n}_1\ \mathsf{n}_2\ (/ \mid //)(\mathsf{o} \mid \mathsf{j} \mid \mathsf{s}) \tag{2.10}$$

2.1.1 Spécification des nœuds dans les XAM

Les XAM ont deux types de nœuds - nœuds élément et nœuds attribut. Chaque nœud est identifié par un identifiant numérique n, et indique pour un élément (ou un attribut) XML quelles sont les informations qu'on peut retrouver dans notre module de stockage, concernant cet élément particulier. Les informations que le stockage peut fournir sont : l'identifiant du nœud pour les nœuds de type élément ($IDSpec$), le nom du nœud ($TSpec$), la valeur de l'élément ou l'attribut ($VSpec$) ou même le contenu textuel de l'élément sérialisé ($CSpec$).

La spécification des identifiants est composée par le symbole ID et un des symboles (i,o,s) selon le niveau des informations que les identifiants renferment. Le premier type d'identifiants (i) spécifie juste le fait qu'on peut identifier de manière unique les éléments stockées. Le symbole o, qui est le type considéré par défaut, spécifie qu'en plus d'une clé unique, l'identifiant reflète aussi l'ordre des éléments dans le document. Le symbole s spécifie que l'identifiant renferme de l'information *structurelle*, permettant par exemple d'inférer des relations structurelles (de type "parent - enfant" et "ancêtre - descendent") entre deux éléments, juste en comparant leurs identifiants.

La spécification des balises est donnée par le symbole Tag ou par une restriction du type [$Tag =$ "$valeur$"]. La première forme indique que le nom de la balise peut être obtenu via le XAM, et la deuxième se comporte comme une sélection, le XAM ayant des informations seulement sur les nœuds étiquetés "valeur". Le contenu textuel des éléments et attributs peut être exprimé d'une manière similaire en utilisant le symbole Val.

Enfin, le langage de description des XAM permet de définir des restrictions d'accès : les informations marquées R (Required) doivent être connues, pour que l'on puisse avoir accès aux information données par le XAM. Cette notation permet aussi de modéliser des stockages utilisant la navigation pour accéder aux données : car, dans ce cas, il faut connaître l'identifiant d'un élément pour accéder à son parent ou à ses enfants etc.

2.1.2 Spécification des arêtes dans les XAM

Les arêtes d'un XAM modélisent les relations structurelles entre les éléments XML ; il y a donc deux types d'arêtes - parent enfant (/) et ancêtre-descendent (//). En plus, sur chaque arête la XAM spécifie la sémantique de jointure (en utilisant les symboles o, j, s pour outerjoin, join et semijoin), nécessaire pour savoir si le XAM peut être utilisé pour répondre à une requête particulière.

2.1.3 Exemples

Dans cette section nous présentons la notation des XAM en nous appuyant sur les exemples de la Figure 2.1. Nous décrivons donc les XAM en utilisant une représentation simple en forme d'arbre. Les nœuds de l'arbre représentent les nœuds du XAM, et les arêtes représentent les arêtes du XAM. Le reste de la spécification est décrite comme des annotations sur les nœuds òu les arêtes.

Le XAM χ_a correspond au cas où le stockage fournit les identifiants de tous les éléments XML stockées, et ces identifiants reflètent l'ordre du document. La description formelle du XAM est la suivante :

$$N = N_E = \{n_1\};\ \ E = \emptyset;\ \ n_1 = 1\ \mathsf{ID}\ \mathsf{o}$$

Nous observons que ce stockage ne peut pas fournir les identifiants des éléments qui, par exemple, ont une étiquette donnée, car le XAM ne mentionne pas les noms des éléments.

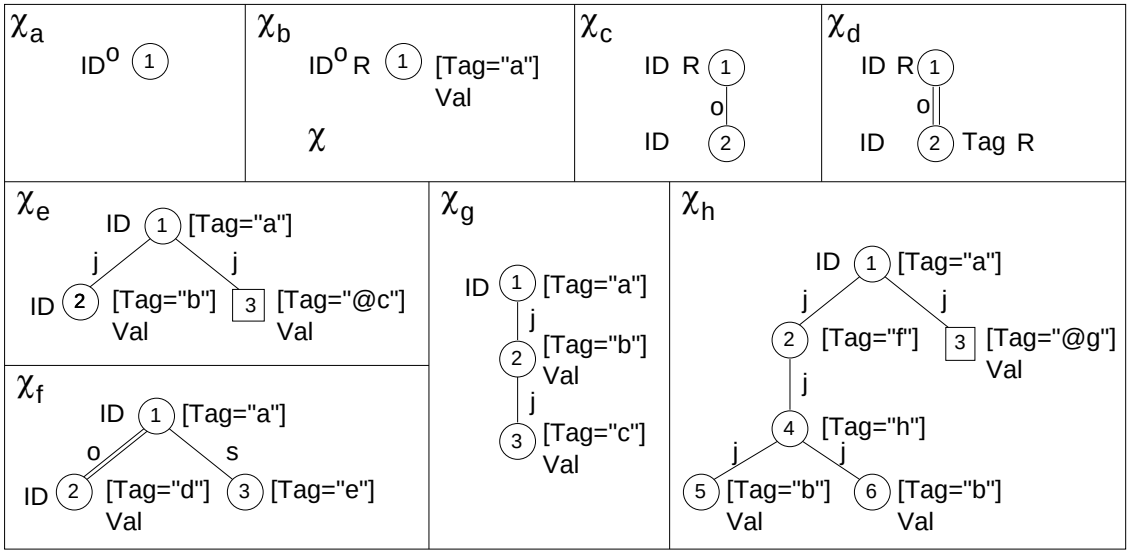

FIG. 2.1 – Exemples des XAM.

Le XAM χ_b nous permet, en utilisant l'identifiant d'un élément étiqueté a, d'obtenir sa valeur textuelle. Ce XAM correspond á un stockage de type "Edge" (présenté en [5]) enrichi avec la restriction d'accès $(ID\ R)$ pour spécifier qu'il faut connaître la valeur de l'identifiant d'un elements pour avoir accès à sa valeur. La description formelle de ce XAM est la suivante :

$$N = N_E \cup N_A;\quad N_E = \{n_1, n_2\}; N_A = \{n_3\}; E = \{(n_1, n_2), (n_1, n_3)\}$$
$$n_1 = 1\ \ \text{ID [Tag="a"]};\qquad n_2 = 2\ \ \text{ID [Tag="b"] Val};\qquad n_3 = 3\ \ \text{[Tag="@c"] Val}$$
$$(n_1, n_2)\ \text{j};\qquad (n_1, n_3)\ \text{j}$$

Le XAM χ_c correspond à un stockage basé sur la navigation, dans lequel, en utilisant l'identifiant d'un élément, nous avons accès aux identifiants de tous ses enfants. Le XAM χ_d correspond au cas dans lequel, étant donné l'étiquette d'un ancêtre et l'étiquette d'un descendant, on peut trouver les identifiants des ancêtres, ensemble avec les identifiants des descendants.

Le XAM χ_e présente un stockage qui permet de trouver, pour tout élément avec l'étiquette a les enfants étiquetés b ainsi que les attributs @c des a, les identifiants des éléments a et b et la valeur textuelle de l'élément b. Nous observons qu'un élément a qui n'as pas un enfant b et un attribut @c ne sera pas fourni par ce XAM, à cause de la sémantique de la jointure.

Le XAM χ_f est similaire, avec les différences que l'arête entre a et d est de type ancêtre-descendent, il a la sémantique de jointure externe, et la relation entre a et ses enfants e a la sémantique de semi-jointure. Ceci signifie que les informations sur un élément a qui n'a pas des descendants d seront présentés dans le XAM. Aussi le XAM va fournir des informations sur des éléments a qui auront au moins un enfant e, mais les informations sur l'enfant e ne seront pas fournies.

Le XAM χ_g fournit les valeurs des éléments b et c pour tous les chaînes a-b-c trouvées dans le document. Nous pouvons aussi obtenir, pour toutes les occurrences du motif d'arbre spécifié, l'identifiant de la racine et les valeurs des feuilles. Nous notons que les deux feuilles à l'extrême gauche partagent la même étiquette b. Ceci, ajouté à la sémantique de jointure des arêtes qui connectent ces feuilles avec leurs enfant b, ont pour effet de stocker le produit cartésien des enfants b.

Dans les sections suivantes, nous allons préférer d'utiliser la représentation graphique des XAM, car elle est plus intuitive et plus facile à lire.

2.2 La sémantique des XAM

Nous définissons maintenant la sémantique des XAM en utilisant le modèle à valeurs complexes ("nested relational"). Ce modèle a été choisi pour son pouvoir d'expression, car il permet d'exprimer plusieurs algèbres (comme l'algèbre relationnelle, l'algèbre relationnelle imbriquée, ou aussi d'autres algèbres d'arbres) utilisées par les moteurs d'exécution de requêtes existants.

2.2.1 Préliminaires

Nous commençons par définir quelques notations utiles : les types associés aux XAM, et quelques fonctions auxiliaires.

Le type associé à un XAM. Pour définir la sémantique des XAM, nous introduisons une famille des schémas dans l'algèbre relationnelle imbriquée, comme suit :

1. Nous considérons un ensembles des *types atomiques* $\mathcal{A}$; dans cette catégorie nous avons les types simple comme "String" , "integer", "double", mais aussi le type atomique des identifiants ID.[1]
2. Nous définissons un ensemble de quatre *attributs élémentaires des nœuds* : ID : ID, Tag : $String$, Val : $String \cup int \cup double$, et Cont : $String$. Pour simplifier, nous allons écrire Val : $\mathcal{A}$ pour faire référence à un type atomique quelconque.
3. A chaque nœud d'un XAM, nous allons associer un ensemble d'*attributs*, qui contient : l'attribut ID si le nœud du XAM a une spécification $IDSpec$, l'attribut Tag si le nœud du XAM a une spécification $TagSpec$, l'attribut Val si le nœud du XAM a une spécification $ValSpec$, et un attribut Cont si le nœud a une spécification $ContSpec$.
4. Étant donné un nœud d'un XAM (nœud étiqueté i), nous définissons *le type associé au nœud* i comme le type complexe Node$i$ dans l'algèbre relationnelle imbriqué, composée par :
   - les attributs du XAM qui correspond au nœud étiqueté i
   - pour tous les enfants j du nœud i, un attribut imbriqué nome Node$j$, de type ensemble sur les types associés aux nœuds étiquetes j.

   Par exemple, le type associé (dans l'algèbre relationnelle imbriquée) au nœud 1 du XAM χ_a de la Figure 2.1 est Node1(ID : ID). Le type associé au nœud 1 du XAM χ_b est : Node1(ID : ID, Tag : $String$, Val : $\mathcal{A}$), le type associé au nœud 2 dans χ_c est Node2(ID : ID), et le type associé au nœud 1 est Node1(ID : ID, Node2 : [(ID : ID)]).
5. Enfin, le *type associé à un XAM* χ est le type associé à la racine du XAM. Par exemple, le type associé à χ_f est :

   Node1(ID : ID, Tag : $String$, Node2 : [(ID : ID, Tag : $String$, Val : $\mathcal{A}$)], Node3 : [(Tag : $String$)])

   Pour simplifier la notation nous pouvons exprimer le même type comme :

   N1(ID, Tag, N2 : [(ID, Tag, Val)], N3 : [(Tag)])

Par la suite cette notation plus simple sera employée pour faire référence aux types associés aux XAM.

[1] Nous pouvons définir plusieurs codages pour les identifiants. Ici nous introduisons le type ID pour faire référence aux identifiants des éléments d'une manière abstraite.

Le nommage des colonnes Nous allons utiliser une syntaxe standard pour les noms des colonnes, en utilisant les points pour indiquer le soustypage comme dans N1.ID, N1.N2.ID, N1.N2.Tag etc. Nous notons que tous les noms de colonnes sont composées par une séquence des noms et un seul attribut final. En plus, nous allons dire qu'une colonne nomme n_1 *est influencée par* une autre colonne nomme n_2 si la séquence de noms de nœud de n_2 est un prefixe de la séquence de noms de nœuds de n_1. Par exemple, N1.N2.Tag est influencée par N1.ID, N1.N2.N3.ID est influencée par N1.N2.Val, N1.N2.ID n'est pas influencée par N1.N3.Tag. Nous allons utiliser cette notions pour définir la sémantique des XAM avec restrictions d'accès (Section 2.2.3).

Les propriétés d'un élément XML Soit e un élément XML. Nous allons désigner l'identifiant persistent associé à e par une schéma de stockage quelconque par e.ID. De même, nous allons utiliser e.Tag, e.Val, e.Cont pour désigner l'étiquette, la valeur, et le contenu textuel de e.

Le contenu d'un élément par rapport á un nœud d'un XAM Soit e un élément XML, et n un nœud d'un XAM. Le contenu de e par rapport à n est un tuple qui a comme signature l'ensemble des attributs correspondent á n, dont les valeurs sont obtenues en utilisant les propriétés du e.

Par exemple, considerons le document D_1 : <root> <a> aaa bbb </root>, et e_r, e_a et e_b étant les éléments root, a et b dans D_1.

Soit n le seul nœud du XAM χ_b dans la Figure 2.1. *Le contenu de e_a par rapport à n* est (ID=e_a.ID, Tag="a", Val="aaa"). Maintenant, soit n le nœud étiqueté "1" dans χ_c; *le contenu de e_a par rapport a n* est (ID=e_a.ID). Le contenu textuel de e_a *par rapport* au nœud 2 de χ_c est le même. Le contenu de e_a *par rapport au nœud 1* de χ_e est : (ID=e_a.ID, Tag="a").

Les prédicats des nœuds Soit t la spécification d'une étiquette comme décrite par le non-terminal $TSpec$, et v la spécification d'une valeur comme décrite par le non-terminal $VSpec$ dans la section 2.1. Soit e un élément XML. Nous définissons deux prédicats :

- $p_t(e) = vrai$ si t contient une spécification Tag=c et e.Tag=c, ou si t ne contient pas une spécification Tag=c. Sinon, $p_t(e) = faux$.
- $p_v(e) = vrai$ si v contient Val=c et e.Val=c, ou si v ne contient pas une spécification Val=c. Sinon, $p_v(e) = faux$.

En utilisant les notations que nous venons d'introduire, nous allons définir la sémantique des XAM en deux étapes. Premièrement, dans la section 2.2.2, nous allons identifier les nœuds d'un document XML qui correspondent à un XAM, en se basant sur les spécifications des nœuds et des arêtes, mais sans utiliser les annotations R. Dans le deuxième pas, nous allons définir la sémantique complète des XAM, en incluant les spécifications R (voir section 2.2.3).

2.2.2 Les informations fournies par un XAM dans l'absence des restrictions d'accès

Nous allons considérer dans une première étape un cas simplifié, dans lequel le XAM χ est ordonné ; nous changerons cela par la suite.

Nous commençons en considérant le cas d'un XAM qui a un seul nœud dans NE et aucune arête. La spécification d'un tel XAM est la suivante : $\chi = (\{n\}, \emptyset)$.

Les informations fournies par χ sur un document d sont organisées comme une liste de tuples imbriqués $[\![\chi]\!]_d$, avec les propriétés suivantes :

1. Le type de $[\![\chi]\!]_d$ est le type dans l'algèbre relationnelle imbriquée qui correspond au seul nœud du XAM n.

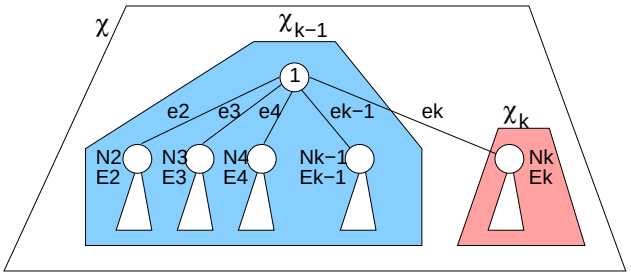

FIG. 2.2 – XAM generique.

2. Nous avons un seul tuple dans $[\![\chi]\!]_d$ pour tout élément e de d avec $p_v(e) = T$ et $p_t(e) = T$. L'ordre des tuples dans $[\![\chi]\!]_d$ respecte l'ordre des éléments dans le document d.
3. Le contenu d'un tuple qui correspond à un élément e est le contenu de l'élément e par rapport à n (suivant les notations de 2.2.1).

Par exemple, nous considérons le document D_1 présenté dans la section precedente et le XAM χ_a dans la Figure 2.1. Nous définissons donc *les informations fournies par le XAM χ_a par rapport au document D_1* comme suit :

$$[\![\chi_a]\!]_{D_1} = \mathsf{N1}\ [\ (\mathsf{ID}= e_r.\mathsf{ID}),\ (\mathsf{ID}= e_a.\mathsf{ID}),\ (\mathsf{ID}= e_b.\mathsf{ID})\ ]$$

Sur le même document, nous considérons le XAM χ_b (défini dans la Figure 2.1). Nous obtenons :

$$[\![\chi_b]\!]_{D_1} = \mathsf{N1}\ [\ (\mathsf{ID}= e_a.\mathsf{ID}\quad \mathsf{Tag}="\mathsf{a}"\ \ \mathsf{Val}="\mathsf{aaa}")\ ]$$

Maintenant, nous considérons le cas général d'un XAM plus complexe. Nous introduisons la définition en s'appuyant sur le XAM décrit dans la Figure 2.2. Nous définissons donc $[\![\chi]\!]$ récursivement, en se basant sur les tuples fournies par χ_{k-1} (identique à χ sauf le dernier sous-arbre et la dernière arête), et celles fournies par χ_k (le dernier sous-arbre connecté à la racine de χ).

Nous faisons une autre hypothèse simplificatrice : soit N1 la racine du XAM χ ; nous considérons dans un premier temps que χ contient une spécification $IDSpec$ pour N1. En conséquence, N1 a un attribut N1.ID dans le type associé au XAM χ (et aussi dans le type associé au χ_{k-1}). En plus, soit Nk la racine de χ_k. Nous considérons aussi que le XAM χ_k contient une spécification $IDSpec$ pour Nk, donc l'attribut Nk.ID apparaîtra dans le type associé au χ_k. Avec ces considérations, pour un document donné d, nous définissons :

$$\begin{aligned}[\![\chi]\!]^j_d = \{&N_1(N_2(\alpha_2), ..., N_{k-1}(\alpha_{k-1}), N_k(\alpha_k))\ tels\ que\\ &((\exists t \in [\![\chi_{k-1}]\!]_d = N_1(N_2(\alpha_2), ..., N_{k-1}(\alpha_{k-1})))\wedge\\ &(\alpha_k = \{\mathrm{t'}\} \mid t' \in [\![\chi_k]\!]_d\ \wedge\ \ t'.N_k.ID \sim t.N_1.ID))\}\end{aligned}$$

$$\begin{aligned}[\![\chi]\!]^s_d = \{&N_1(N_2(\alpha_2), ..., N_{k-1}(\alpha_{k-1}))\ tels\ que\\ &((\exists t \in [\![\chi_{k-1}]\!]_d = N_1(N_2(\alpha_2), ..., N_{k-1}(\alpha_{k-1})))\wedge\\ &(\alpha_k = \{\mathrm{t'}\} \mid t' \in [\![\chi_k]\!]_d\ \wedge\ \ t'.N_k.ID \sim t.N_1.ID))\}\end{aligned}$$

$$
\begin{aligned}
[\![\chi]\!]^o_d = \{&N_1(N_2(\alpha_2), ..., N_{k-1}(\alpha_{k-1})) \; tels \; que \\
&((\exists t \in [\![\chi_{k-1}]\!]_d = N_1(N_2(\alpha_2), ..., N_{k-1}(\alpha_{k-1})))\wedge \\
&(\alpha_k = \{\text{t'}\} \mid t' \in [\![\chi_k]\!]_d \; \wedge \; t'.N_k.ID \sim t.N_1.ID))\} \cup \\
&\{N_1(N_2(\alpha_2), ..., N_{k-1}(\alpha_{k-1}), N_k(\emptyset)) \; tels \; que \\
&\neg((\exists t \in [\![\chi_{k-1}]\!]_d = N_1(N_2(\alpha_2), ..., N_{k-1}(\alpha_{k-1})))\wedge \\
&(\alpha_k = \{\text{t'}\} \mid t' \in [\![\chi_k]\!]_d \; \wedge \; t'.N_k.ID \sim t.N_1.ID))\}
\end{aligned}
$$

Dans les définitions ci-dessous, $\alpha_1, \alpha_2, \ldots, \alpha_k$ désigne des valeurs quelconques des attributs des tuples imbriqués

Nous notons le fait que l'indice supérieur de $[\![\chi]\!]_d$ correspond à la sémantique de la relation structurelle de e_k (*j-jointure*, *s-semi-jointure* et *o-outerjoin*). Aussi, la notation $\sim$ dans $t'.N_k.ID \sim t.N_1.ID$ est un raccourci pour le type de relation structurelle décrite par e_k (ancêtre - descendant ou parent - enfant). Pour les arêtes de type *outerjoin* nous introduisons la notation $N_k(\emptyset)$ pour désigner un tuple du type composé N_k, et dans lequel toutes les valeurs atomique sont *null*.

Maintenant nous renonçons à l'hypothèse sur la présence des identifiants, et nous permettons l'absence des attributs ID pour les nœuds racine du χ_{k-1} et χ_k.

Soit χ'_{k-1} et χ'_k les XAM obtenus en ajoutant les attributs ID correspondant aux XAM χ_{k-1} et χ_k, et soit χ' le XAM obtenu par la composition des XAM χ_{k-1} et χ_k, comme dans la figure 2.2. Nous avons donc les définitions suivantes :

$$
\begin{aligned}
[\![\chi]\!]^j_d &= \pi_{\text{type}(\chi)}[\![\chi']\!] \\
[\![\chi]\!]^s_d &= \pi_{\text{type}(\chi)}[\![\chi']\!]^s_d \\
[\![\chi]\!]^o_d &= \pi_{\text{type}(\chi)}[\![\chi']\!]^o_d
\end{aligned}
$$

où la projection $\pi_{\text{type}(\chi)}$ garde seulement les attributs qui correspondent au type associé au XAM χ en éliminant de cette manière les attributs ID de χ'_{k-1} et χ'_k qui ont ete introduits pour définir la composition de χ'_{k-1} et χ'_k.

L'exemple suivant illustre la définition récursive de l'ensemble des tuples fournies par un XAM, en utilisant les règles définies précédemment.

Exemple : La sémantique des XAM sans restrictions d'accès

Étant donné le document D_2 :

```
<root> <a @c="c1"> <b> b1 </b> <b> b2 </b> </a>
       <a @c="c2"> </a>
       <a @c="c3"> <b> b3 </b> <a>
</root>
```

Pour définir la sémantique du XAM χ_e de la figure 2 de D_2, nous commençons par définir les XAM élémentaires : le XAM χ^1_e est composé seulement par le nœud étiqueté 1 dans χ_e. De manière similaire, χ^2_e et χ^3_e sont composés seulement par les nœuds étiquetés 2, et 3 dans χ_e. Après, nous définissons le XAM $\chi^{1,2}_e$ qui est constitué seulement par les nœuds 1 et 2 de χ_e, et l'arête qui les

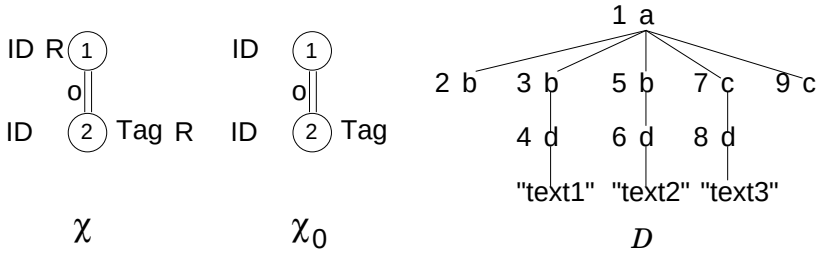

FIG. 2.3 – Le XAM avec des restrictions d'accès χ, le XAM χ_0 obtenu suite à l'élimination des restrictions d'accès, le document stocké D.

unit. Nous avons donc :

$[\![\chi_e^1]\!]_{D_2}$ = N1 [(ID=a_1.ID, Tag="a"), (ID=a_2.ID, Tag="a"), (ID=a_3.ID, Tag="a")]

$[\![\chi_e^2]\!]_{D_2}$ = N2 [(ID=b_1.ID, Tag="b", Val="b1"), (ID=b_2.ID, Tag="b", Val="b2"), (ID=b_3.ID, Tag="b", Val="b3")]

$[\![\chi_e^3]\!]_{D_2}$ = N3 [(Tag="@c", Val="c1"), (Tag="@c", Val="c2"), (Tag="@c", Val="c3")]

$[\![\chi_e^{1,2}]\!]_{D_2} = [\![\chi_e^1]\!]_{D_2} \otimes [\![\chi_e^2]\!]_{D_2} =$
= N1 [(ID=a_1.ID, Tag="a", N2 [(ID=b_1.ID, Tag="b", Val="b1"), (ID=b_2.ID, Tag="b", Val="b2")]),
(ID=a_3.ID, Tag="a", N2 [(ID=b_3.ID, Tag="b", Val="b3")])]

$[\![\chi_e]\!]_{D_2} = [\![\chi^{1,2}]\!]_{D_2} \otimes [\![\chi_e^3]\!]_{D_2} =$
= N1 [(ID=a_1.ID, Tag="a", N2 [(ID=b_1.ID, Tag="b", Val="b1"), (ID=b_2.ID, Tag="b", Val="b2")],
N3 [(Tag="@c", Val="c1")]),
(ID=a_3.ID, Tag="a", N2 [(ID=b_3.ID, Tag="b", Val="b3")], N3 [(Tag="@c", Val="c3")])]

2.2.3 Les informations fournies par un XAM dans la présence des restrictions d'accès

Les informations stockées par un module décrit par un XAM peuvent être parfois inaccessibles. Les valeurs de certaines informations peuvent être requises pour accéder au contenu stocké. Deux exemples typiques sont : la sécurité de données (une clé doit être connue pour accéder aux enregistrements) ou dans le cas des index. Ces restrictions d'accès sont définies par la notation R dans la spécification des XAM (section 2.1).

Soit un XAM et $a_1, a_2, \ldots a_k$ les noms de colonnes de $[\![\chi]\!]$ qui correspondent aux colonnes marquées R dans χ. Soit T_R un ensemble des *tuples de "bindings" pour* χ : un ensemble des tuples qui ont les colonnes $b_1, b_2, \ldots b_k$, les mêmes types et imbrication que la restriction de $[\![\chi]\!]$ à $a_1, a_2, \ldots a_k$.

Exemple : La sémantique des XAM dans la présence des restrictions d'accès

Soit le XAM χ de la Figure 2.3 (gauche) qui a été introduit dans la Figure 2.1(d). Au milieu de la Figure 2.3 il y a le XAM χ_0 obtenu en enlevant les restrictions d'accès de χ. Finalement, à droite dans la figure 2.3, nous avons un document XML auquel nous pouvons accéder par les XAM χ et χ_0.

Algorithme 1 : Étant donné un tuple de $[\![\chi_0]\!]$, et un tuple de "bindings" pour les valeurs requises, fournit les informations accessibles de $[\![\chi]\!]$ avec ces "bindings".

input : t (tuple de $[\![\chi_0]\!]$)
input : t_R (tuple de T_R, composé des colonnes $b_1, b_2, \ldots, b_k$)
output : t_2 (tuple de $[\![\chi]\!]$)

```
1  t2 ← t∅ //un tuple avec le même type (signature) que t, rempli avec des valeurs ∅ dans
   tous les champs atomiques
2  bm ← 0 // bitmap de m bits, ou m est le nombre des attributs dans [[χ]]
3  //filtre les éléments de t en utilisant tous les conditions du "bindings"
4  foreach ai in a1, a2, ..., ak do
5      trouve la valeur de t2.ai, comme l'intersection des valeurs de t.ai avec des valeurs de
       tR.bi
6      if t2.ai = ∅ then
7          exit.//failure
8      else
9          bm[ai]←1
10         foreach attribute a'i of t, a'i influencées par ai do
11             calcule la valeur de t2.a'i en filtrant les valeurs t2.ai par t.a'i
12             bm[a'i]←1
13 foreach attribute a''i dans [[χ]] tel que bm[a''i]=0 do
14     //copie les attributs qui n'apparaissent pas dans les bindings
15     copie t1.a''i dans t2.a''i
```

Les tuples fournies par χ_0 :
([N1 : (ID : 1, [N2 : (ID : 2, Tag :b), N2 : (ID :3, Tag :b), N2 : (ID :5, Tag :b), N2 : (ID :7, Tag :c), N2 : (ID :9, Tag :c)])], [N1 : (ID : 2, [N2 : (∅)])], [N1 : (ID : 3, [N2 : (ID : 4, Tag :d)])], [N1 : (ID : 4, [N2 : (∅)])], [N1 : (ID : 5, [N2 : (ID : 6, Tag :d)])], [N1 : (ID : 6, [N2 : (∅)])], [N1 : (ID : 7, [N2 : (ID : 8, Tag :d)])], [N1 : (ID : 8, [N2 : (∅)])], [N1 : (ID : 9, [N2 : (∅)])])
Les "bindings" :
T_R=([N1 : (ID :1, [N2 : (Tag : [b, c])])], [N1 : (ID :5, [N2 : (Tag :[d])])])
Les tuples fournies par le XAM χ dans la présence des restrictions d'accès T_R :
([N1 : (ID : 1, [N2 : (ID : 2, Tag :b), N2 : (ID :3, Tag :b), N2 : (ID :5, Tag :b), N2 : (ID :7, Tag :c), N2 : (ID :9, Tag :c)])], [N1 : (ID : 5, [N2 : (ID : 6, Tag :d)])])

TAB. 2.1 – Les tuples fournies par le XAM χ_0 de la figure 2.3 (en haut) ; les "bindings" des restrictions d'accès T_R (au milieu) ; tuples fournies par le XAM χ de la figure 2.3 dans la présence des restrictions d'accès T_R (en bas).

Le tableau 2.1 présente les tuples fournies par ces XAM. Premièrement, nous considérons les tuples fournies par le XAM χ_0. Chaque tel tuple a deux attributs nommes N1 et N2 ; les attributs des tuples fournies par le XAM sont étiquetés en concordance avec les numéros des nœuds du XAM N1, N2, N3 etc. Chaque attribute peut être :

- un tuple avec un ou plusieurs attributs parmi ID, Val, Tag, et Content. Par exemple, la valeur de N1 dans le premier tuple fourni par χ_0 est (ID : 1).
- un ensemble (ou liste) de tuples qui ont le même type (signature). Par exemple, la valeur de N2 dans le même tuple est une liste des tuples de (ID : 2, Tag : b) jusqu à (ID : 9, Tag : c).

Nous avons un tuple fourni par χ_0 pour un nœud de D, car le nœud étiqueté 1 dans χ_0 peut correspondre à chaque nœud de D. En plus, à cause de la sémantique de semi jointure sur l'arête, dans χ_0 chaque tuple met ensemble un nœud avec la liste de tous ses descendants ; ces liste peuvent être vide dans le cas des éléments sans enfants (comme les éléments 2, 4 etc).

2.3 Modéliser des stockages et des index en utilisant les XAM

Dans cette section, nous allons démontrer que notre formalisme est assez général pour modéliser beaucoup des systèmes de stockage XML ainsi que beaucoup d'index existants. Nous commençons par modéliser des techniques de stockage très utilises comme les modèles DOM [21] et "Tag Partition" [8]. Nous ne donnérons pas beaucoup des détails sur ces techniques, en nous concentrant plutôt sur l'accès au structures stockées que nous modélisons en utilisant les XAM.

2.3.1 Modéliser les stockages XML basses sur des systèmes relationnels

Suite à la maturité des système relationnels et du support (aussi théorique et logiciel) du cet modèle, nombreux stockages XML se basant sur le stockage dans des basses de données relationnelles ont été proposés. Tous ces stockages utilisent les clefs externes pour capturer les relations structurelles de type parent-enfant ou ancêtre-descendant, et des valeurs ordinales pour capturer la relations d'ordre entre les enfants de mêmes parents.

Les stockages indépendants du schéma du document XML Une première catégorie de stockages relationnels qui utilise ces principes est constitue par les stockages XML qui *n'utilisent pas d'informations sur les schémas* des documents stockées. Nous allons donc présenter les approches : *Edge*, *Basic*, et *Universal* [5], *Shared* et *Hybrid* [18], *XRel* [11], et *XParent* [9] et nous allons voir comment les modéliser en utilisant les XAM.

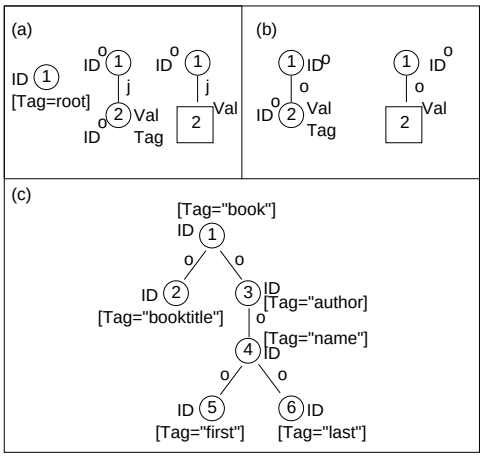

FIG. 2.4 – Les XAM pour les approches *Edge* (a), *Binary* (b), et *Universal table* (c).

L'approche Edge : est basée sur le stockage des tous les arcs parent-enfant dans une table relationnelle avec le schéma suivant :

$$Edge(source, target, ordinal, name, flag)$$

ou *source* et *target* identifient les nœuds unis par l'arc, *name* est l'étiquette sur l'arc, et *ordinal* est un numéro utilisé pour exprimer l'ordre local des nœuds[2]. A chaque nœud du document XML est associé un identifiant unique et les valeurs des attributs et des éléments de type texte sont stockées dans des tables séparées ayant le schéma $V[vid, value]$. Les XAM n'exprime pas d'une manière explicite l'ordre local, mais celle-ci peut être derivée à partir des identifiants qui reflètent l'ordre (marquées par ID o dans la spécification des nœuds). Nous notons que dans l'approche *Edge*, l'arbre XML est étiqueté sur les arêtes, mais il peut être facilement transformé en un arbre étiqueté sur les nœuds. La description de l'approche *Edge* est présentée dans la Figure 2.4(a). Le premier XAM modélise l'accès aux éléments XML et le deuxième l'accès aux attributs. Les auteurs de *Edge* proposent aussi un index sur la colonne *source*, qui peut être décrit en rajoutant une spécification ID R dans notre notation.

[2]Par ordre local des nœuds, nous entendons l'ordre parmi tous les enfants d'un nœud, qui ont le même étiquetté. Par exemple, le premier élément étiqueté author enfant d'un élément étiqueté book, le deuxième enfant author et ainsi de suite

FIG. 2.5 – Exemple de document XML utilisé pour modéliser des stockages

L'approche Binary est basée sur le groupement des edges qui ont la même étiquette dans la même table relationelle (ceci équivaut à partition horizontale de *Edge* en utilisant la colonne *name* comme attribut de partitionnement) :

$$B_{name}(source, ordinal, flag, target)$$

Pour accéder aux éléments stockées, nous devons maintenant connaître le nom de la table qui contient les informations sur l'élément cherché à partir de son nom. Nous pouvons modéliser cette approche en ajoutant une spécification [Tag=val] dans le XAM présenté pour *Edge* (Figure 2.4(c)).

L'approche Universal utilise une seule table pour stocker tous les arêtes. La table utilise a donc le schéma suivant :

$$Universal(source, ordinal_{n1}, flag_{n1}, target_{n1}, \ldots, ordinal_{nk}, flag_{nk}, target_{nk})$$

ou $n_1, \ldots, n_k$ sont les étiquettés présentes dans le document. Étant donné le fait que cette table est définie formellement dans [5] comme la jointure externe complète de toutes les tables *Edge*, le XAM correspondant est celui décrit dans Figure 2.4(c).

Modéliser les stockage relationnelles qui utilisent les informations sur le schéma Une autre approche proposée pour le stockages des documents XML est d'utiliser la structure des DTD pour stocker les données XML dans des tables relationnelles. Dans cette catégorie, nous trouvons les approches *Basic*, *Shared* et *Hybrid*, qui ont été evaluées dans [18] et aussi *XRel* [11] et *XParent* [9].

Dans [18], la décision de créer une nouvelle table pour un élément ou d'insérer les informations qui le concerne dans la table de son parent est prise selon le fait que l'élément est partage ou non par d'autre éléments dans le DTD. Parce qu'un élément peut être inséré dans plusieurs éléments qui le référencent *Basic*, *Shared* et *Hybrid* diffèrent par le degré de redondance supporté. Par exemple, dans l'approche *Basic*, nous insérons autant des descendants d'un élément que possible dans la table des éléments qui le référence, mais pourtant des relations pour tous les éléments sont crées car les arbres XML peuvent être enracinés au niveau de chaque élément. La modélisation des stockages par des XAM peut être ilustreée sur le document de la figure 2.5 ; les schémas des tables utilisées sont

présentés dans la Figure 2.6(a). Chaque table relationnelle est modelisée par un XAM, comme le montre la Figure 2.6(b).

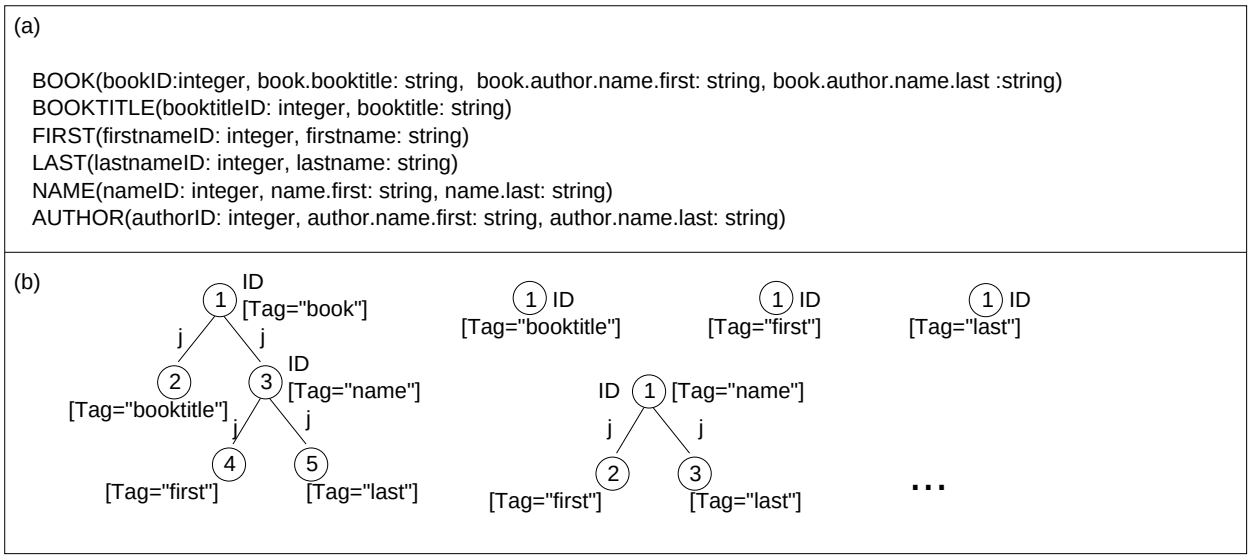

FIG. 2.6 – Les XAM qui modèle le stockage *Basic*. Les tables relationnelles crées (a), et les XAM correspondants (b).

Les approches utilisées par *XRel* [11] et *XParent* [9] se basent sur le principe de stocker aussi des informations sur les chemins dans le RDBMS. Nous mentionnons que ces stockages sont similaires du point de vue de l'accès aux données aux approches du type *Path Partition* (présente dans la section suivante) donc les XAM qui les modélisent sont les mêmes. Dans cette approche, les contenu des nœuds XML, et des pointeurs vers leurs chemins correspondants (en utilisant des clés externes) sont stockés ensemble (dans la même table relationnelle). Les schémas des relations stockées par *XRel* sont les suivantes :

$Element(docID, pathID, start, end, index, reindex)$
$Attribute(docID, pathID, start, end, value)$
$Text(docID, pathID, start, end, value)$
$Path(pathID, pathexp)$

Les XAM qui modélisent ce type de stockage sont les mêmes que ceux présentés dans la Figure 2.8.

2.3.2 Modéliser des stockages XML natives

Beaucoup des modèles de stockages pour XML ne se basent pas sur le modèle relationnel. Dans cette section nous allons évaluer le pouvoir expressif des XAM par rapport à ces modèles. La Figure 2.7 ilustre les descriptions des modèles de stockage natifs les plus fréquemment utilisés.

Le modèle DOM [21] est plus une interface de programmation d'application basé sur des arbres utilisées pour manipuler des données XML, qu'un modèle de stockage ; ce modèle permet pourtant de décrire des méthodes d'accès simples aux nœuds des arbres XML. Nous discutons ici ce modèle parce que il est utilisé par de nombreux systèmes XML comme interface pour accéder aux données [19, 17, 15, 16].

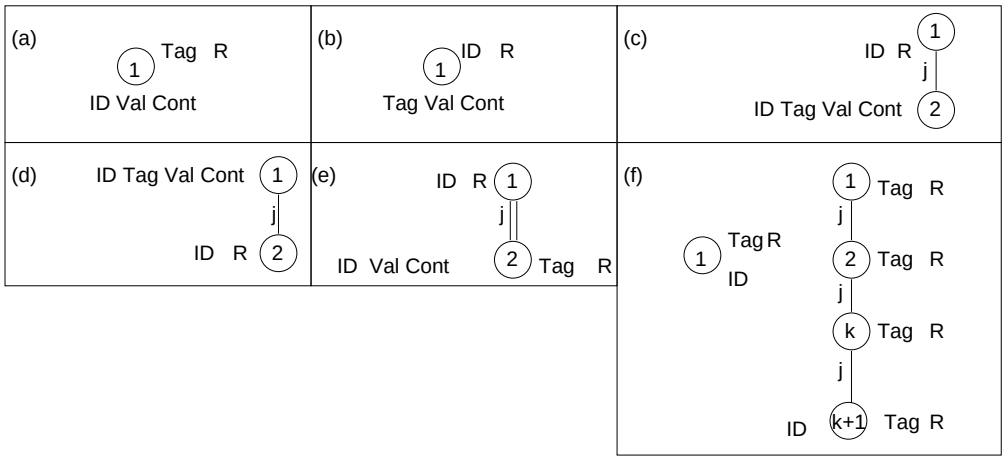

FIG. 2.7 – Les XAM pour DOM(a)-(e) et "Path partitioning" (f).

Le modèle DOM fournit : l'accès aux enfants d'un nœud, l'accès aux nœuds qui ont une étiquette donnée et la navigations parmi les enfants de mêmes parents. Pour modéliser ce type d'accès par les XAM, nous assimilons un pointeur au nœud DOM (de type élément) à l'identifiant unique du nœud du XAM. Les XAM permettent la modélisation de ce modèles d'accès, en exceptant la navigation directe parmi les enfants de mêmes parents (par exemple, l'accès fournit par l'axe "following sibling" du DOM). Par exemple, la primitive DOM $GetElementsByTagName$ fournit l'accès aux ensembles des éléments, étant donnée la valeur de leurs étiquettes. Le XAM qui est utilisé pour décrire ce type d'accès est présenté dans la figure 2.7(a).

Le modèle DOM permet aussi la navigation parent-enfant et enfant-parent (par les primitives $getParentNode$ et $getChildNodes$) ; nous décrivons ce type de navigation par un XAM composé de deux nœuds (un pour l'enfant et l'autre pour parent) Figure 2.7(c) et Figure 2.7(d). Enfin, le modèle DOM permet aussi l'accès aux descendants d'un nœud si l'identifiant de l'ancêtre et l'étiquetté du descendant sont connues. Ce type d'accès peut être modélisé par le XAM présenté dans la figure 2.7(e).

Modéliser des stockages basés sur "Tag partitioning" (TP) Les stockages du type "TP" sont utilisés dans beaucoup des systèmes comme Timber [7, 8] et Natix [10]. Ces systèmes utilisent le partitionnement des documents XML basse sur l'étiquette des éléments et l'utilisation de l'étiquette comme un clé pour trouver tous les éléments ayant une telle etiquette. Du point de vue de l'accès au données, l'effet du partitionnement est très similaire aux primitives DOM $GetElementsByTagName$, et les mêmes XAM peuvent être utilisées pour les modéliser (Figure 2.7(a)).
Les systèmes basses sur Path Partitioning - (PP) [1] partitionne les données selon les *chemins* trouvées dans le document. En plus, le système Gex [1, 13] utilise des identifiants structurels et organise les données dans des séquences ordonnés. En utilisant cette approche pour tous les chemins de forme $/tag_1/tag_2/\ldots/tag_k$, ou tag_1 est la racine du document XML, $k \geq 1$ et tous les nœuds sur le chemin sont connectés par (/), nous pouvons accéder aux identifiants des éléments ayant le étiquetté tag_k. D'une manière similaire, pour tous les chemins $/tag_1/tag_2/.../tag_k/\#PCData$ ou $/tag_1/tag_2/.../tag_k/@attr$ (ou $attr$ est un nom d'attribut), nous pouvons trouver tous les paires $(ID, value)$ ou $value$ est le contenu textuel de l'élément tag_k dans le cas de $\#PCData$), ou la valeur de l'attribut (dans le cas de $@attrName$). L'accès aux identifiants des nœuds, étant donné un chemin, est modélisé par le XAM en figure 2.7(f).

Alternatives pour modéliser des stockage PP en utilisant les XAM Nous notons qu'en modélisant des stockages PP, deux approches distinctes existent ayant différents degrés de généralité :

- Les XAM présentés dans la Figure 2.7(f). Cette approche utilise un seul XAM pour chaque *longueur* de chemin jusqu'à la profondeur maximale du document XML. Cette approche est très compacte, car le stockage est décrit par peu de XAM.
- Une approche plus précise est d'utiliser le filtrage des éléments par la spécification [Tag=value] plutôt que le filtrage plus général Tag R. Cette approche utilise un XAM pour chaque chemin présent dans le document. Nous allons préférer cette dernière description, à cause du gain de précision au niveau du filtrage offert par le XAM (Figure 2.8).

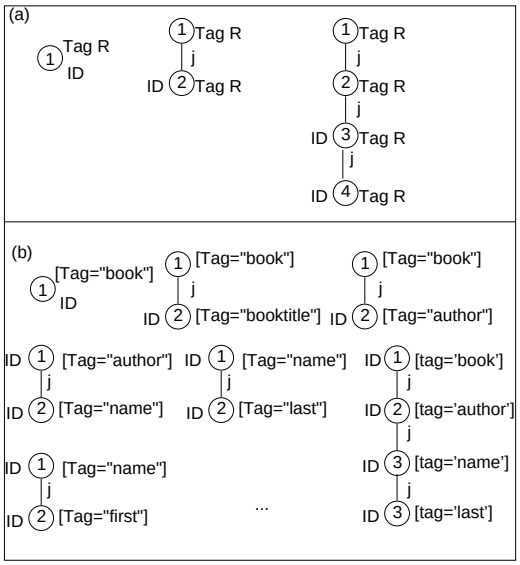

FIG. 2.8 – Modéliser les stockages basés sur Path Partitioning : a)une représentation compacte, b)la représentation conseillée.

2.3.3 Modéliser des index par des XAM

Des nombreuses techniques d'indexations ont ete considérées par les systèmes de gestion de données XML. Cependant, comme nous sommes intéressés juste de voir comment nous pouvons modéliser l'accès aux données indexées, nous allons omettre les détails des différentes techniques d'indexation et nous allons les grouper dans trois catégories, en se basant sur les structures utilisées dans l'indexation.

Les méthodes d'indexation bassées sur les nœuds Les premières méthodes d'indexation pour XML ont considéré les éléments/attributs comme unité de base d'indexation ; les expressions de chemin complexes ont été décomposées en collections d'expressions simples. Seulement l'accès par des expressions simples (éléments ou attributs) est accéleré par les indexes, d'une manière similaire à la primitive *getElementByTag* du modèle *DOM*.

L'un des systèmes qui utilisent cette méthode est *XISS* [12] qui propose une schéma de numérotation, une extension d'identifiants structurels qui permet les mises à jour. Les structures indexées dans *XISS* sont :

- *un index sur les éléments* - étant donné une étiquette T, l'index sur les éléments fournit la liste des éléments qui ont la même étiquette ;

- *un index sur les attributs* - étant donné un nom d'attribut A, cet index fournit tous les attributs ayant le même nom A.
- *un index structurel* - étant donné un élément/attribut cet index fournit le parent de l'élément ainsi que les éléments/attributs enfants.
- *un index des noms* - pour accélérer l'accès aux chaînes des caractères (utilisés dans le contenu textuel des éléments et attributs, et aussi dans les noms des éléments/attributs) XISS n'utilise aucune comparaison des caractères ; au lieu de cela, l'index des noms (implanté comme un arbre B) fournit pour toutes les chaînes de caractères s toutes les paires s,$name/stringID$ qui seront utilisées plus tard dans les traitements.
- *un index des valeurs* - cet index est similaire aux *index des noms*, mais cette fois sont indexées aussi les valeurs d'autre types.

Les XAM qui décrivent les index de XISS sont présentées dans la Figure 2.9. Nous notons que l'index des noms n'est pas explicitement exprimé par un XAM.

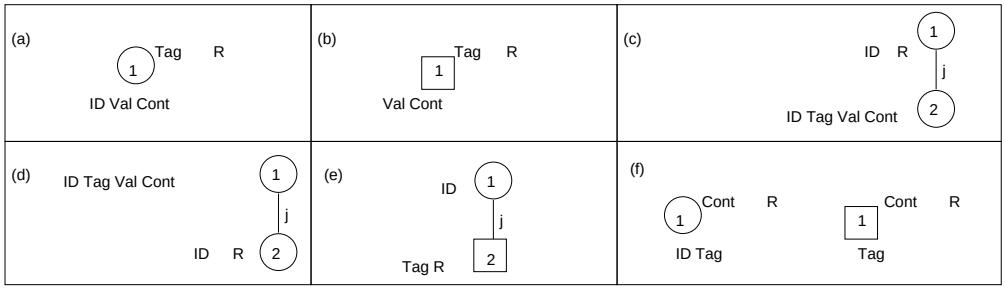

FIG. 2.9 – Les indexes du XISS : l'index des éléments (a), l'index des attributs (b), l'index structural (c-e), l'index des valeurs (f).

Les méthodes d'indexation basées sur les chemins sont une autre catégorie de méthodes d'indexation òu les chemins sont les unités d'indexation de base. Dans cette approche, le moteur d'indexation fournit des sommaires sur la structure des chemins, qui seront utilisées par le moteur d'exécution de requêtes pour accelerer la traversée des données XML.

Les *DataGuides* [6] et les *1-indexes* [14] font partie de la catégorie des indexes sur les chemins qui représentent tout les chemins de la racine. Leur fonctionnalité, du point de vue de l'accès aux données, est similaire à la fonctionnalite présentée pour les méthodes basées sur "path partition". Les XAM sont donc aussi similaires à ceux présentés dans la Figure 2.8.

Le *Template Index (T-Index)* [14] est un généralisation du DataGuide, permettant un compromis flexible entre espace/niveau de généralité. En utilisant des templates comme le suivant :

$$select\ x\ from\ t = T_1 x_1 T_2 ... T_n x_n,$$

où chaque T_i est une expression du chemin ou une formule d'expressions du chemins, le T-index donne accès au sous-arbres qui ont la racine $x_1 \ldots x_n$ et qui vérifient le template. Par exemple, pour trouver tous les livres qui ont un auteur dont le nom commence par "Suciu", nous pouvons écrire la requête suivante :

$$select\ x\ from\ t = (*.book)x(name/last[val = "Suciu"])$$

Un *T-index* nous donnéra l'accès rapide aux bindings conformes aux template modélisé par le XAM dans la Figure 2.10

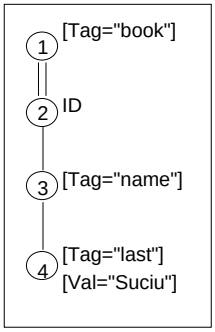

FIG. 2.10 – Un *T-index* pour une requête simple.

D'autres approches utilisent seulement/aussi des index sur les chemins les plus fréquemments utilisées (des fois même les sous-arbres les plus fréquemment utilisées). *IndexFabric* [4] et *APEX* [3] sont deux exemples en ce genre.

Index Fabric code les chemins comme des chaînes de caractères et utilise un index spécial pour les chaînes de caractères, une structure similaire à un *Patricia trie*, utilisée souvent pour retrouver des motifs dans les chaînes de caractères. Dans cette approche, deux types de chemins sont indexés :

- **Raw paths** sont des chemins racine-feuille qui existent tels quels dans les données ; ces chemins sont codées en utilisant pour chaque nom d'élément un désignateur (prefix codé dans un identifiant - chaîne de caractères). L'index représenté par le *Patricia trie* est donc similaire aux stockages "path partitioning" du point de vue de l'accès aux données ; il est donc modélisé par les même XAM (Figure 2.11(b)).
- **Refined paths** - sont des chemins spécifiques du données qui sont reordonées pour optimiser certaines requêtes. En utilisant des informations sur le jeux de requêtes les plus souvent utilisées des motifs d'arbre fréquents sont identifiés et indexés d'une manière similaire aux *raw paths*. Par exemple, pour une requête qui cherche les livres qui ont ete écrites par X et Y ensemble, un index comme celui dans la Figure 2.11(c) peut être utilisé.

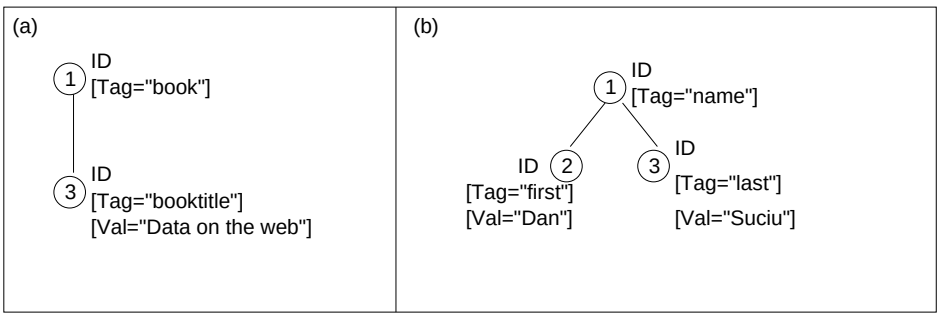

FIG. 2.11 – Index Fabric : Les indexes "raw path" (a), et "refined path" (b).

APEX [3] est une technique d'indexation similaire aux *refined path* indexes, utilisant seulement les chemins les plus fréquemment traversés pour accélerer l'exécution des requêtes. En revanche, *APEX* utilise des algorithmes de "data-mining" pour créer un sommaire pour chemins les plus utilisés. Aussi *APEX* garantit l'indexation des chemins de longueur deux. Les XAM qui modélise cette technique d'indexation sont similaires aux ceux présentés pour les "refined paths" de (*Index*

Fabric Figure 2.11(c)).

Les méthodes d'indexation basées sur les motifs d'arbres Pour éviter la décomposition de la requête dans des sous-requêtes élémentaires, les accélerer et puis les "coller" par de jointures certains systèmes construisent leurs index à base de structures d'arbre. Une technique d'indexation de ce type est implantée dans ViST [20]. D'une manière similaire au *Index Fabric*, *ViST* est basée sur les *refined paths*. En revanche d'utiliser des informations sur des jeux de requête pour déterminer les multi-chemins les plus fréquents des requêtes, dans *ViST* les documents XML et les requêtes sont codées dans une représentation séquentielle commune. En utilisant cette approche, exécuter des requêtes sur ces documents se réduit à chercher des sous-chaînes entre les deux entités codées (les documents XML et les requêtes). Du point de vue de l'accès aux données, *ViST* est identique à un index sur les "refined paths" pour tous les requêtes multi-chemins avec des enfants nommes diffèremment à chaque niveau (similaire aux XAM du figure 2.11(c))

2.4 Restrictions sur les XAM

Comme nous avons deja observé, les XAM ne modélisent pas :

- l'accès basé sur les axes sibling ou accès direct aux premier/dernier enfant d'un élément XML.
- des stockage qui permetent de trouver "tous les éléments ⟨a⟩ qui ont au moins/exactemment 2 éléments ⟨b⟩ comme enfants".
- les stockages qui permetent de trouver "tous les éléments ⟨a⟩ qui n'ont aucun enfant ⟨b⟩ ".

Nous ne modélisons pas l'axe "sibling" car il conduit à des stratégies d'evaluation de requêtes basés sur la navigation, et donc assez inefficaces. En ce qui concerne l'incorporation des contraintes numériques, et de la négation (absence d'un fils ayant une certaine étiquette), nous choisissons de les ommetre pour le moment, car leur inclusion compliquerait le processus d'identification des XAM utiles pour une requête donnée.

Chapitre 3

Optimisation de requêtes XQuery sur les XAM

Nous avons commencé l'étude de l'optimisation de requêtes XQuery en utilisant notre langage de modélisation de stockages. Nous avons conçu un algorithme générique qui, étant donné un ensemble des XAM X, une requête XQuery Q sur un document D, et un ensemble optionnel de contraintes C satisfaites par D, trouve tous les sous-ensembles de X, qui sont suffisants pour répondre à la requête Q. Pour chaque tel ensemble, l'algorithme trouve un ensemble d'opérateurs algébriques (sélection, projection...) qui adaptent chaque XAM, en gardant juste les parties nécessaires pour répondre à Q. Enfin, l'algorithme trouve un plan d'exécution logique pour Q, basé sur les parties trouvées au pas antérieur, et des opérateurs génériques de jointure. Le but de cette section est de présenter cet algorithme.

3.1 Préliminaires

3.1.1 XQuery

XQuery [22] est le langage recommandé par W3C pour l'interrogation des données XML. Les structures de base dans XQuery sont les expressions FLWOR. La sémantique d'une expression FLOWR simple peut etre présentée intuitivement par une requête simple sur le document de la figure 2.5 :

```
FOR $x in /bib
FOR $y in $x/book
WHERE $y/author/@id="suciu"
RETURN $y/title
```

L'evaluation de XQuery se fait en trois étapes.

- La phase de "binding" - la clause FOR génère une séquence d'éléments *book* qui apparaissent sous l'élément *bib* et lie la variable x à chaque élément. Apres, la deuxième clause FOR pour chaque "binding" de x lie la variable y aux enfants de l'élément. Le résultat de cette phase est une liste des paires de x et y.
- La phase de filtrage - la clause $WHERE$ filtre les éléments obtenus dans la première phase en utilisant le prédicat $\$y/author/@id = "suciu"$. Nous obtiendrons seulement les éléments de type *book* qui ont l'identifiant de l'un des auteurs *suciu*.

– La phase de reconstruction du résultat - la clause *RETURN* groupe les résultats obtenus et construit des éléments XML comme résultat. Dans notre exemple la clause *RETURN* est très simple - on obtient seulement les éléments *title* comme ils apparaissent dans le document.

3.1.2 Le fragment de XQuery utilisé

Nos algorithmes peuvent être facilement étendus pour les utiliser pour répondre à des requêtes XQuery complexes. Nous allons restreindre notre exposition à un fragment de XQuery. La grammaire du fragment de XQuery considéré est similaire à celle considérée par [2].

(1) FLWR : := (ForClause|LetClause)+ WhereClause ReturnClause
(2) ForClause : := FOR $\$f_{v1}$ IN E_1,...,$\$f_{vn}$ IN E_n
(3) LetClause : := LET $\$l_{v1}$:= E_1,...,$\$l_{vn}$:= E_n
(4) WhereClause : := WHERE $\varphi(E_1, ...E_n)$
(5) ReturnClause : := RETURN $E_1...E_n$
(6) E_i : := FLWR | XPATH

Dans la suite nous allons considérer des requêtes XQuery comme étant seulement des expressions FLWR non imbriquées et nous allons omettre les autres détails (fonctions, la partie procedurale de XQuery etc.). Nous allons écrire ces requêtes en utilisant une forme d'arbre similaire au "Generalized Tree Patterns" présenté dans [2].

3.2 Répondre a des requêtes XQuery en utilisant les XAM

3.2.1 Algorithme de re-écriture des requêtes XQuery en utilisant les XAM

Cette section présente l'algorithme qui, étant donné un ensemble des XAM décrivant un stockage physique, une requête XQuery et des informations sur la structure du document stocké (comme celles obtenues à partir d'un DataGuide [6]) trouve toutes les re-écritures de la requête en utilisant (peut-être seulement une partie de) l'ensemble des XAM.

La structure principale de l'algorithme est la suivante :

Sommaire de l'algorithme de re écriture :
Input : un ensemble des XAM X=$\{X_1X_2...X_n\}$ décrivant un stockage physique, une requête XQuery mise en forme d'arbre TPQ comme présente dans [2]
Output : un ensemble des plans d'exécutions qui utilisent les XAM pour répondre à la requête Q

I. Pour tous les nœuds de la requête Q, trouver tous les XAM (ou combinaisons de XAM) qui sont nécessaires et suffisantes pour fournir les informations sur le nœud de Q.

II. Calculer un coût pour les combinaisons trouvées pour chaque nœud de la requête, et ordonner les alternatives en utilisant le coût trouvé.

III. Trouver tous les plans d'exécution (QEP) qui répondent à Q en combinant les XAM trouvées au pas I, en utilisant les combinaisons les moins couteauses d'abord.

IV. Optimiser davantage les plans d'exécutions trouvés au pas précédent.

Remarque : les etapes II-IV peuvent être intercalées pour finaliser des plans d'exécution au fur et à mesure qu'ils sont obtenus.

Nous allons dècrire maintenant en détail les étapes présentés précédement. Nous n'allons pas détailler le dernier pas de l'algorithme car les optimisations des plans d'exécutions basés sur les

XAM sont similaires aux optimisations classiques (eliminer les parties qui ne sont pas nécessaires des plans, descendre les sélections et projections etc.)

Pas I. Trouver le sous-ensemble des XAM relevant pour chaque nœud de la requête

Pour trouver les XAM qui ont des informations sur les nœuds de la requête, nous utilisons la notion de *compatibilité de nœuds*. Intuitivement, un nœud de la requête est compatible avec un nœud d'un XAM si le nœud du XAM contient d'une manière implicite ou explicite le nom du nœud de la requête.

Formellement, *un nœud n étiquéte l de la requête Q est compatible avec un nœud m d'un XAM χ* ssi dans la spécification de m nous trouvons une des spécifications suivantes :

- une spécification $[Tag = l]$.
- une spécification generique Tag .

Pour utiliser un XAM pour obtenir des informations sur un certain nœud, nous avons des fois besoin de faire des modifications d'abord. Un exemple typique des ces modifications est la sélection qui doit être appliquée pour filtrer une spécification générique de type Tag. Nous allons nommer l'ensemble de toutes les modifications que doivent être faites à un (ensemble de) XAM pour donnér des informations sur un certain nœud de la requête, un *Edit Script*. Il faut remarquer que les *Edit Scripts* fournissent seulement les modifications qui sont nécessaires pour trouver des informations concernant un certain nœud de la requête, mais n'appliquent pas d'autres optimisations, car celles-ci seront décidées sur un plan complet.

Pour avoir accès au contenu d'un XAM il faut parfois fournir au XAM certaines valeurs (celles marquées dans la spécification par R). L'algorithme essaie d'utiliser dans ce but les valeurs fournies par la requête, et de combiner des autres XAM pour trouver les valeurs demandées.

Dans ce premier pas de l'algorithme, nous utilisons trois ensembles XS, ES et RS :

- XS contient tous les XAM qui doivent être utilisés pour obtenir des informations concernant un certain nœud
- ES contient tous les *Edit Scripts* nécessaires pour transformer les nœuds nécessaires
- RS contient toutes les valeurs requises ou les champs des XAM qui peuvent fournir ces valeurs.

Nous associons à chaque XAM X une liste des nœuds qui sont couverts par X (X peut être utilisé pour fournir des informations sur chaque nœud de cette liste). Nous allons nommer ces listes $CoverLists$.

À la fin du premier pas de l'algorithme, nous construisons un tableau ($CandXAMs$) qui donne, pour chaque nœud de la requête n les alternatives pour trouver les informations concernant n sous la forme des triplets (XS, ES, RS). Ce premier pas est présenté dans l'algorithme 2.

Pas II. Estimer le coût des alternatives de re-écriture et trier les alternatives dans l'ordre croissant des coûts.

Ce deuxième pas de l'algorithme a le rôle de trier l'espace des alternatives permettant de choisir une re-écriture pour un nœud de la requête. Un modèle de coût plus avancé doit être considérer, mais pour le but de ce stage, nous avons utilisé des heuristiques plus simples. Ainsi, nous avons considéré le coût comme une fonction du nombre des attributs présents dans le XAM, et du nombre des nœuds de la requête couvert par une certaine alternative.

Pas III. La construction des plans d'exécution

L'algorithme pour construire les plans d'exécution est présenté dans la suite. Les plans d'exécution de requêtes sont construits en collant les XAM qui donnent des informations sur les nœuds de

Algorithme 2 : Répondre á des requêtes en utilisant les XAM : Pas I. Trouver le sous-ensemble des XAM relevants pour chaque nœud de la requête

input : l'ensemble des XAM qui décrit un stockage physique X
: la requête Q
output : l'ensemble des alternatives dans la re-écriture des nœuds $CandXAM$

```
1  foreach q nœud de Q do
2      foreach XAM X_i de X do
3          newAlternative=faux
4          XS ← ∅; ES ← ∅; RS ← ∅
5          foreach nœud n de X_i do
6              if nodeCompatible (q,n) then
7                  if X_i ne contient pas spécifications R then
8                      calculer l'Edit Script associé - ES_i
9                      XS ← X_i
10                     ES ← ES_i
11                     newAlternative←vrai
12                     ajouter q à la CoverList de X_i
13                 else if toutes les valeurs r_i requises sont en Q then
14                     calculer l'Edit Script associé - ES_i
15                     XS ← X_i
16                     ES ← ES_i
17                     RS ← r_i
18                     newAlternative←vrai
19                     ajouter q à la CoverList de X_i
20                 else
21                     r_i^1=les valeurs dans Q marquées R dans X_i
22                     trouver récursivement toutes les combinaisons des valeurs R de Q qui
                       peuvent être fournies par d'autres XAM (X_s) - r_i^2
23                     if pour chaque valeur R il y a au moins une valeur then
24                         calculer les ES associés aux X_s
25                         combiner les valeurs r_i^1 et r_i^2 dans un tuple de "bindings" r
26                         mettre à jour XS =X_s ∪ X_i, ES, RS = {r}, CoverLists
27                         newAlternative←vrai
28             if newAlternative=vrai then
29                 CandXAMs[q].add(XS,ES,RS)
30     if CandXAMs[q]=∅ then
31         exit ;//je ne peux pas répondre à la requête Q en utilisant seulement X
```

la requête, dans le but de couvrir l'arbre de la requête.

Dans le premier pas de l'algorithme de re-écriture nous vérifions la compatibilité entre les nœuds de la requête et les nœuds du XAM en se basant seulement sur les étiquettes des nœuds. Pourtant, pour trouver les correspondances entre les nœuds de la requête et les XAM, il faut tenir compte des informations sur le contexte (chemin) dans lequel les nœuds apparaissent. Nous allons utiliser la compatibilité entre une arête de la requête et une arête (ou un chemin) d'un XAM, tirer profit du contexte d'un nœud. Nous allons nomme cette relation de compatibilité *compatibilité des chemins*.

La compatibilité des chemins Nous exprimons le *contexte d'un nœud d'un XAM* en utilisant des expressions XPath dèsignant le chemin d'un certain nœud du XAM. Le contexte implicite du nœud racine d'un XAM est //. Les informations sur la structure des documents stockés peuvent être utilisées pour détailler les contextes des nœuds.

Par exemple, pour le XAM Xf dans la figure 2.1 le contexte du nœud 2 est $//a[/e]//d$. Cette expression peut être détaillée d'avantage, en utilisant les informations fournies par le DataGuide (par exemple, les chemins comportant des // peuvent être remplecés par les unions des chemins élémentaires trouvés dans les données, en s'appuyant sur un DataGuide).

Par extension, nous définissons le *contexte d'un nœud de la requête* comme l'expression XPath qui donné le chemin vers le nœud, augmentée avec les conditions structurelles imposées en plus par la requête.

En utilisant les notions du contexte de nœuds le problème de *compatibilité des chemins* se reduit à une problème d'inclusion d'expressions XPath :

Un nœud q d'une requête XQuery Q est dans la relations de compatibilité de chemins avec un nœud n d'un XAM X ssi :

- le nœud q est *(nœud) compatible avec le nœud n* **et**
- *le chemin XPath qui décrit le contexte du nœud q dans Q est inclus dans le chemin XPath qui décrit le nœud n dans X*

Algorithme 3 : Répondre à des requêtes en utilisant les XAM : Pas III. La construction des plans d'exécution

---

input : l'ensemble des XAM qui décrit un stockage physique X
: la requête Q
: l'ensemble des alternatives dans la re-écriture des nœuds $CandXAM$
: structure similaire à un DataGuide, contenant des informations sur la structure des documents stockés D

output : la liste de tous les plans d'exécutions $Solutions$ (on associe un plan d'exécution à un ensemble des re-écritures, une pour chaque nœud)

```
1 Solutions ← ∅
2 foreach alternative a_i de Q.root dans CandXAM do
3 |   partialSolution ← generateQepForNode (Q.root,a_i)
4 |   generateQepRec (X, Q, CandXAMs, partialSolution, Q.root, D)
5 |   if partialSolution! = ∅ then Solutions ← Solutions ∪ partialSolution
6 return Solutions
```

Pour mettre en correspondence les arêtes de la requête avec des arêtes/chemins dans les XAM, nous utilisons des informations sur la structure des documents XML chargées dans le stockage, informations qui ne sont données par un résume structural similaire à un DataGuide.

L'algorithme 3 est basé sur une traversée récursive de l'arbre de la requête. Nous commençons par le nœud racine et, nous vérifions si la solution partielle trouvée couvre un nœud enfant/descendant

et l'arête qui lie l'enfant à son parent. Si oui, nous marquons que le nœud enfant/descendant à été couvert par un XAM existant dans la solution partielle et nous continuons en profondeur. Si la solution partielle ne couvre pas le nouveau nœud et/ou l'arête correspondente, nous essayons de trouver des XAM parmi les alternatives pour le nouveau nœud trouvées dans le pas I de l'algorithme, qui peuvent être "collées" avec la solution partielle par une jointure structurelle. Pour pouvoir combiner le nouveau XAM et la solution partielle, nous devons avoir des identifiants structurels aux deux extremités de l'arête qui lie le parent et l'enfant (ou l'ancêtre et le descendant). Si c'est le cas, nous ajoutons les nouveaux XAM, et modifions la solution partielle en ajoutant le plan d'exécution pour le nouveau nœud (obtenu à partir de l'alternative choisie) qui lie les deux et continuons en profondeur. Sinon, l'algorithme revient en arrière. Nous utilisons une fonction *generateQepForNode* qui recoit en paramètre un nœud de la requête et une alternative associée dans CandXAM et retourne le plan d'exécution pour obtenir le nœud de la requête (ce plan d'exécution est juste le résultat de l'applications des edit-scripts).

Fonction `generateQepRec`$(X, Q, CandXAMs, Soutions, currentNode, D)$ construit récursivement tous les plans d'exécution

input : l'ensemble des XAM qui décrit un stockage physique X
: la requête Q
: l'ensemble des alternatives dans la re-écriture des nœuds $CandXAM$
: structure similaire à un DataGuide contenant des informations sur la structure des documents stockés D
: le nœud courant de Q : $currentNode$

output : la liste partielle des plans d'exécution $Solutions$

```
7  foreach solution partielle s_i dans S do
8      foreach enfant p_j de currentNode do
9          existXAMCovered ← faux
10         foreach nœud x de s_i compatible avec p_i do
11             existXAMCovered ← vrai
12             s_i.coversList=s_i.coversLists ∪ p_i
13             mettre à jour s_i(XS,ES,RS)
14             generateQepRec (X, Q, CandXAMs, PartialSolutions, p_i, D)
15         if existXAMCovered = faux then
16             //essayer de coller un nouveau XAM à s_i
17             if currentNode n'a pas une spécification ID then
18                 éliminer s_i de Solutions
19             else
20                 found1Sol=faux
21                 foreach a_i re-écriture de p_i do
22                     if a_i contient une spécification ID then
23                         coller un nouveau XAM
24                         s_i=structuralJoin(s_i,constrQepForNode(p_i))
25                         found1Sol=vrai, mettre à jour s_i(XS,ES,RS)
26                         generateQepRec (X, Q, CandXAMs, PartialSolutions, p_i)
27                 if found1Sol=faux then
28                     exit ;//je ne peux pas répondre à la requête Q en utilisant seulement X
```

Chapitre 4

Conclusions

Nous avons introduit les XAM comme un modèle générique de représentation de stockages persistants des données XML. Nous y avons aussi inclu des caractéristiques spécifiques aux systèmes de stockage XML, telles que les identifiants persistants, et les restrictions d'accès. Ce modèle a été validé par rapport à un ensemble important des techniques de stockage et indexation qui ont été modélisées en utilisent notre langage. Nous avons également conçu une première version d'algorithme qui utilise les XAMs pour répondre à des requêtes XQuery.

4.1 Perspectives

Dans une première étape nous allons raffiner l'algorithme d'optimisation des requêtes, en utilisant les XAM, et nous étudierons en détail les proprietes de l'algorithme. Notamment, il faudrait prouver la correction et la complétude de l'algorithme de re-écriture.

Il faudrait aussi s'intéresser à la possibilité d'utiliser des informations sur le jeu de requêtes et sur la structure des documents pour proposer des stockages.

Nous allons continuer d'implanter l'algorithme de re-écriture des requêtes dans un optimiseur de XQuery et nous comparerons les performances de cet optimiseur avec les performances des autres systèmes récents, pour mesurer le prix payé pour la généricité du stockage.

Nous étudierons aussi la possibilité d'étendre le fragment XQuery utilisé, puis nous étudierons les problèmes spécifiques liées à l'adaptation de notre algorithme à d'autre langages de requêtes.

Le but à long terme de ce travail est d'explorer la construction d'un optimiseur, qui soit générique aussi au niveau du langage de requêtes.

Table des figures

2.1 Exemples des XAM. 8
2.2 XAM generique. 11
2.3 Le XAM avec des restrictions d'accès χ, le XAM χ_0 obtenu suite à l'élimination des restrictions d'accès, le document stocké D. 13
2.4 Les XAM pour les approches *Edge* (a), *Binary* (b), et *Universal table* (c). 16
2.5 Exemple de document XML utilisé pour modéliser des stockages 17
2.6 Les XAM qui modèle le stockage *Basic*. Les tables relationnelles crées (a), et les XAM correspondants (b). 18
2.7 Les XAM pour DOM(a)-(e) et "Path partitioning" (f). 19
2.8 Modéliser les stockages basés sur Path Partitioning : a)une représentation compacte, b)la représentation conseillée. 20
2.9 Les indexes du XISS : l'index des éléments (a), l'index des attributs (b), l'index structural (c-e), l'index des valeurs (f). 21
2.10 Un *T-index* pour une requête simple. 22
2.11 Index Fabric : Les indexes "raw path" (a), et "refined path" (b). 22

Bibliographie

[1] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. Efficient query evaluation over compressed XML data. In *EDBT*, 2004.

[2] Z. Chen, H.V. Jagadish, L. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns : On efficient evaluation of XQuery. In *VLDB*, 2003.

[3] C.-W. Chung, J.-K. Min, and K. Shim. Apex : an adaptive path index for XML data. In *SIGMOD*, 2002.

[4] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A fast index for semi-structured data. In *VLDB*, pages 341–350, 2001.

[5] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. In *IEEE Data Engineering Bulletin*, 1999.

[6] R. Goldman and J. Widom. Dataguides : Enabling query formulation and optimization in semistructured databases. In *VLDB*, Athens, Greece, 1997.

[7] H. Jagadish, S. Al-Khalifa, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, and Y. Wu. Timber : A native XML database. Technical report, University of Michigan, http ://www.eecs.umich.edu/db/timber/, April 2002.

[8] H. V. Jagadish, Shurug Al-Khalifa, Laks Lakshmanan, Andrew Nierman, Stylianos Paparizos, Jignesh Patel, Divesh Srivastava, , and Yuqing Wu. Timber : A native XML database. In *SIGMOD*, page 672, 2003.

[9] H. Jiang, H. Lu, W. Wang, and J. Xu. XParent : An efficient RDBMS-based XML database system. In *ICDE*, 2002.

[10] C.C. Kanne and G. Moerkotte. Efficient storage of XML data. In *ICDE*, 2000.

[11] D. Kha, M. Yoshikawa, and S. Uemura. An XML indexing structure with relative region coordinate. In *ICDE*, pages 313–320, 2001.

[12] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.

[13] I. Manolescu, A. Arion, A. Bonifati, and A. Pugliese. Path sequence-based XML query processing. Submitted for publication, 2004.

[14] T. Milo and D. Suciu. Index structures for path expressions. *Lecture Notes in Computer Science*, 1540 :277–295, 1999.

[15] The Qexo system. http ://www.gnu.org/software/qexo/.

[16] The QizX system. www.xfra.net/qizxopen/.

[17] A. Sahuguet. The Kweelt system. http ://sourceforge.net/projects/kweelt.

[18] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents : Limitations and opportunities. In *VLDB*, 1999.

[19] J. Simeon. The Galax system. http ://db.bell-labs.com/galax.

[20] H. Wang, S. Park, W. Fan, and P. Yu. ViST : A dynamic index method for querying XML data by tree structures. In *SIGMOD*, pages 110–121, 2003.

[21] Document Object Model. http ://www.w3.org/DOM/.

[22] The XQuery language. www.w3.org/TR/xquery, 2003.