Un modèle de stockage de données XML basé sur les séquences

Ioana Manolescu[*] — Andrei Arion[*]
Angela Bonifati[**] — Andrea Pugliese[***]

[*] *INRIA Futurs, projet Gemo*
4, rue Jacques Monod
F-91893 Orsay cedex

{prénom.nom@inria.fr}

[**] *ICAR CNR*
Via P. Bucci 41/C
I-87036 Rende (CS)

bonifati@icar.cnr.it

[***] *DEIS*
University of Calabria
Via P. Bucci 41/C
I-87036 Rende (CS)

apugliese@deis.unical.it

RÉSUMÉ. *Nous présentons le modèle de stockage XML basé sur les chemins du document. Ce modèle sépare la structure du contenu du document, et utilise des séquences comme structures logiques et physiques de stockage. Il permet un accès sélectif aux données et la combinaison efficace de plans d'accès, en s'appuyant sur un résumé structurel des chemins. Ce modèle a été mis en œuvre dans un prototype de système de gestion de bases de données XML. Nous démontrons son intérêt par une série d'expériences.*

ABSTRACT. *We present the path sequence storage model, a new logical model for storing XML documents. This model partitions XML data and content according to the document paths; and uses ordered sequences as logical and physical structures. Its main advantages are: fast and selective data access plans, and intelligent combination of such plans, based on a summary of the document paths. This model has been fully implemented within an XML database prototype; we demonstrate its advantages through extensive experiments.*

MOTS-CLÉS : *optimisation de requêtes XML, modèle de stockage, jointures structurelles.*

KEYWORDS: *XML query optimization, storage model, structural joins.*

1. Introduction

Le traitement de données XML donne actuellement lieu à de nombreux travaux de recherche. Le language XQuery [XQU04] standardisé au sein du W3C approche de sa forme finale ; de nombreuses recherches sont actuellement concentrées sur les techniques de stockage et traitement de requêtes XQuery [ALK 02, WU 03, TEU 03, BRU 02, HAL 03, CHE 03, MAR 03, NGO 03]. Nous illustrons les étapes essentielles dans le traitement d'une requête XML sur l'exemple fourni par le document XML de la figure 2(a). Ce document correspond à un site d'enchères on-line, comprenant des données sur les clients, les objets à vendre aux enchères, ainsi qu'une classification de ces objets en catégories.

Exemple 1. Considérons la requête :

```
Q1  for     $i in //asia//item, $d in $i/description
    where   $i//keyword="romantic"
    return  <gift>
                  <name> {$i/name} </name>
                  {$d//emph}
            </gift>
```

Pour chaque objet à vendre en Asie et chaque description d'un tel objet, tel qu'à l'objet est associé au moins un descendant keyword ayant la valeur *romantic*, cette requête construit un nouvel élément gift contenant le nom de l'objet, ainsi que les parties de sa description contenues dans des éléments emph. Des objets comprenant le mot-clé *romantic* plusieurs fois doivent produire un seul élément gift, avec son nom et ses éventuels descendants emph.

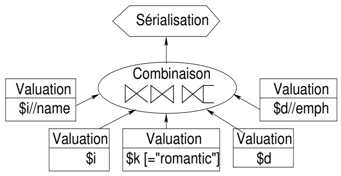


Figure 1. *Stratégie générale d'évaluation de XQuery*

Une partie importante de l'évaluation de requêtes XQuery consiste à trouver des *valuations des variables*. Les valuations d'une variable v comprennent la liste des éléments (sans doublons du point de vue de l'identité des nœuds et dans l'ordre du document) atteints en suivant l'expression de chemin définissant v. Par exemple, les valuations de la variable $i sont les éléments trouvés en évaluant l'expression de chemin //asia//item. De même, les valuations de la variable $d sont obtenues en suivant le chemin //description à partir de chaque valuation de la variable $i ; les valuations de $d doivent aussi être uniques et refléter l'ordre du document.

Une stratégie générale d'évaluation de requêtes XQuery procède dans les étapes successives, esquissées dans la figure 1 :

(i) Trouver des valuations pour toutes les variables, ainsi que pour les sous-requêtes contenues dans la clause return. *(ii) Combiner* ces valuations à l'aide des jointures. Enfin, *sérialiser* les résultats XML, suivant les composantes de la clause return. Si les étapes précédentes manipulent directement des identifiants d'éléments XML, cette dernière étape exige de récupérer et étiqueter le contenu des éléments.

Le plan dans la figure 1 illustre cette approche pour la requête de l'exemple 1. Nous omettons dans cette figure les détails de chaque étape ; il seront donnés par la suite. Nous introduisons la variable supplémentaire $k, correspondant au mot-clé qui doit figurer dans la description. Les jointures dans l'étape de combinaison associent des valuations des variables selon des prédicats de jointure par valeur (absents dans cet exemple), ou par des *relations structurelles* entre les identifiants des éléments présents dans ces valuations. Cette dernière classe de jointures est connue sous le nom de *jointures structurelles* [ALK 02, CHI 02, BRU 02, ZHA 01]. Dans la figure 1, à l'intérieur de l'étape de combinaison, les valuations de $i et de $d sont combinées par une *jointure structurelle* ; en même temps, les valuations de $i et $k sont combinées par une *semi-jointure structurelle*, puisque les objets ayant plusieurs mots-clés satisfaisant le critère de recherche doivent produire seulement un résultat ; enfin, des *jointures externes* sur des critères structurels connectent les valuations de $i/name et $d//emph aux valuations de $i et $d, puisqu'il faut produire un résultat même pour des objets sans nom et/ou sans descendants de type emph.

L'efficacité de l'évaluation de requêtes XQuery dépend de l'efficacité de ces trois étapes : la valuation, la combinaison et la sérialisation des résultats. Cette dernière étape peut être effectuée de manière très efficace, une fois que le contenu à sérialiser a été récupéré et trié dans le bon ordre [SHA 00]. En conséquence, la performance est en fait déterminée par : la valuation des variables, basée sur les prédicats de la requête et les relations entre variables et la combinaison de ces valuations.

Des travaux récents ont fourni des techniques efficaces pour la mise en œuvre des jointures structurelles [ALK 02] et pour leur ordonnancement [WU 03, CHE 03], en s'appuyant sur un stockage relationnel ou à base d'arbre, enrichi avec des index [HAL 03, CHE 03]. Ces index permettent d'accéder aux *identifiants des éléments ayant une certaine étiquette*. Nous appellons cette approche de stockage et indexation *partitionnement par l'étiquette* ou PE.

Cet article présente un nouveau modèle de stockage XML à base de *séquences de chemins* ou PC. Ce modèle sépare la structure du contenu d'un document XML, en suivant les chemins de ce document et stocke le document comme un ensemble de *séquences ordonnées*. Son principal avantage réside dans son efficacité lors de la valuations des variables, qui peut être plus rapide de plusieurs ordres de grandeurs que dans le cas de PE. Cet avantage est dû à la connaissance structurelle précise de ce document, qui est incluse dans le modèle PC. Notre travail apporte les contributions suivantes :

– Nous décrivons le modèle logique et physique de stockage de données XML basé sur les *séquences de chemins*. Ce modèle est plus compact que PE et permet le chargement rapide des documents. Il comporte un *résumé structurel* du document XML (de type DataGuide [GOL 97]), qui guide l'évaluation des requêtes.

– Nous montrons qu'en s'appuyant sur le résumé de chemins, la valuation des variables est bien plus efficace que dans le modèle PE.

– Nous étendons le modèle d'exécution basé sur des itérateurs, pour les opérateurs dont les résultats sont dans l'ordre du document. En utilisant des informations sur le résumé des chemins et en s'appuyant sur cette extension, nous dérivons une famille de jointures structurelles, qui utilisent des prédicats sélectifs sur l'une des deux entrées de la jointure pour accéder à une partie seulement de l'autre entrée.

– Nous montrons que notre modèle de stockage et de traitement de requêtes s'intègre bien dans les techniques d'optimisation de XQuery basées sur PE [WU 03, CHE 03]. En plus de ses avantages pour la valuation de variables, notre modèle fournit des informations structurelles précises pour l'optimiseur, ce qui permet d'améliorer encore plus l'efficacité de l'évaluation de requêtes XQuery.

L'organisation de cet article est la suivante. La section 2 présente les travaux existants, et en particulier le modèle le plus populaire actuellement basé sur le *partitionnement à base d'étiquette* (TP), qui est très proche de notre travail. La section 3 décrit le modèle de stockage basé sur les séquences de chemin. Nous expliquons par la suite comment les requêtes XQuery peuvent être traitées de manière efficace en s'appuyant sur ce modèle. La section 4 décrit la construction des plans de valuation des variables. En section 5, nous traitons de la combinaison des plans de valuation, en s'appuyant sur le résumé de chemins. La section 6 présente nos résultats expérimentaux. Nous présentons notre conclusion dans la section 7.

2. Etat de l'art en traitement de requêtes XML

Notre travail se situe dans le contexte du traitement de requêtes XQuery basé sur des jointures structurelles. Nous présentons ici brièvement les travaux les plus similaires, et discutons les différences entre notre approche et ces travaux.

Stockage partitionné par étiquette (PE). Les travaux les plus proches des nôtres ont été faits dans le cadre des systèmes Timber [ALK 02, WU 03, BRU 02, CHE 03], Niagara [HAL 03] et Natix [FIE 02]. Ces systèmes stockent des documents XML en deux types de structures différentes. D'abord, un arbre persistant est stocké sur disque, reflétant tout le contenu du document et permettant la navigation directe d'un nœud à un autre. En plus de cet arbre, un index *partitionné par l'étiquette des nœuds* permet d'accéder aux identifiants structurels de tous les éléments dans le document ayant une certaine étiquette.

Valuation des variables sur un stockage PE. Du point de vue du traitement de requêtes, la différence la plus importante entre PE et PC est dû à l'étape de valuation des variables. Examinons d'abord la valuation des variables sur un stockage PE. Considérons le cas où nous devons trouver les valuations d'une variable liée à un chemin simple, comme dans l'exemple suivant[1] :

$$\mathsf{Q}_2 \quad \mathsf{for} \quad \mathsf{\$v\ in\ //l_1//l_2//\ldots//l_k} \ldots$$

Pour trouver les valuations de \$v, plusieurs étapes sont nécessaires. (i) Accéder à l'index partitionné par étiquette successivement avec les clés d'indexation l_1, l_2, et jusqu'à l_k, afin de récupérer les identifiants des éléments ayant ces étiquettes. (ii) Combiner les k ensembles d'identifiants ainsi obtenus, soit par $k-1$ jointures structurelles [WU 03], soit par une jointure unique à branches multiples [BRU 02]. (iii) Une étape de tri et une élimination de doublons peuvent s'avérer nécessaire, afin de s'assurer que les valuations sont obtenues dans l'ordre du document et sans doublons. Il faut remarquer ici que les jointures structurelles peuvent avoir détruit l'ordre des entrées, et introduit des doublons.

Chacune de ces étapes peut introduire des problèmes de performance. La première étape ci-dessus peut conduire à lire plus d'identifiants d'éléments que ceux qui ont une chance de contribuer aux jointures. C'est le cas, par exemple, si certains éléments étiquetés l_2 ne sont pas des descendants d'éléments étiquetés l_1, et donc ne peuvent pas contribuer des valuations pour \$v.

En conséquence, l'étape (ii) peut conduire à un effort de jointure plus grand que nécessaire. Si une jointure à entrées multiples est utilisée [BRU 02], l'effort dû aux jointures inutiles est réduit, mais ne peut pas être complètement éliminé, tandis que les accès aux données inutiles ne sont pas affectés. Enfin, l'étape (iii) ci-dessus peut introduire un effort significatif, puisque l'élimination des doublons et le tri sont des opérations coûteuses (et bloquantes).

Considérons maintenant le cas d'une requête plus complexe de type *motif d'arbre*, correspondant, par exemple, à une expression XPath complexe ou à un fragment XQuery plus complexe, comme dans l'exemple suivant :

$$\begin{array}{lll}
\mathsf{Q}_3 & \mathsf{for} & \mathsf{\$v\ in\ //l_1//l_2//\ldots//l_{n_1}}, \\
 & & \mathsf{\$u\ in\ \$v//l_{n_1+1}//\ldots//l_{n_2}} \\
 & & \ldots \\
 & & \mathsf{\$x\ in\ \$v//l_{n_2+1}//\ldots//l_n} \\
 & \mathsf{where} & C(\mathsf{\$v, \$u, \ldots, \$x}) \\
 & \mathsf{return} & \ldots
\end{array}$$

Dans ce cas, par un raisonnement similaire, le partitionnement par étiquette entraîne la lecture des identifiants de tous les éléments étiquetés l_1, l_2, ..., l_n, de les combiner par $n-1$ jointures structurelles, et peut-être d'appliquer aussi des étapes de tri et d'élimination des doublons. Les problèmes de performance dûs à la lecture

1. Dans cet exemple et dans la suite, nous utilisons des chemins séparés par // ; l'analyse est très similaire dans le cas ou les étapes sont séparées par /.

de données inutiles, à l'effort de jointure trop élevé et aux éliminations de doublons inutiles restent les mêmes.

Le modèle de stockage partitionné par les chemins (PC), que nous introduisons dans la suite, applique un partitionnement vertical basé sur les chemins du document ; ceci permet d'éviter certaines lectures de données inutiles et réduit l'effort des jointures, de tri et d'élimination de doublons, illustrés dans les exemples ci-dessus.

Sur un stockage organisé par chemins, nous pouvons traiter des requêtes de chemin linéaires, telles que Q_2, en lisant directement les identifiants nécessaires, sans évaluer aucune jointure structurelle et sans avoir besoin de tris et d'éliminations de doublons. Des requêtes plus complexes, telles que Q_3, nécessitent toutefois un nombre de jointures structurelles égal au nombre d'expressions de chemin linéaires impliquées[2], mais leurs entrées seront plus petites qu'en utilisant PE.

L'effet de la minimisation à base de schéma. On peut se demander si la performance de PE en ce qui concerne la valuation ne pourrait pas être améliorée par une analyse statique des expressions de chemin associées aux variables. L'information sur les schémas peut en effet être utilisée pour simplifier ou minimiser une requête, c'est-à-dire, l'amener à une forme équivalente mais syntactiquement plus simple..

Par exemple, considérons la requête Q_4 : $//\mathsf{l}_1//\mathsf{l}_2$. Si on peut inférer à partir du schéma que tout élément l_2 a un ancêtre l_1, alors Q_4 peut être écrite de manière équivalente : $\mathsf{Q'}_4$: $//\mathsf{l}_2$. Si le stockage est basé sur PE, alors l'évaluation de $\mathsf{Q'}_4$ pourrait être bien plus facile que l'évaluation de Q_4, puisque $\mathsf{Q'}_4$ lit uniquement les identifiants nécessaires, ne nécessite pas de jointure, pas de tri et pas d'élimination de doublons.

En général, étant donnée une requête de chemin simple Q, plusieurs requêtes Q^1, $\mathsf{Q}^2, \ldots, \mathsf{Q}^k$ peuvent être équivalentes à Q et avoir des performances très différentes sur un stockage PE. Nous appelons *partitionnement par étiquette optimisé* (PEO) la stratégie d'évaluation de requêtes qui, pour une requête q, énumère toutes les formes équivalentes de q et choisit celle qui présente les meilleures performances sur un stockage PE. Ainsi, PEO utilise l'ordonnancement de jointures structurelles le moins coûteux pour la requête donnée, et n'applique des tris et des éliminations de doublons que si ces opérations sont nécessaires. Intuitivement, PEO est le meilleur plan d'exécution possible sur un stockage PE.

Nous constatons que les performances de PC dépassent non seulement celles de PE, mais aussi celles d'OPE. La raison en est l'accès sélectif aux données qui est fourni par PC. Par la suite, cet accès sélectif est dû au fait qu'un stockage partitionné par chemin codifie des informations complexes sur la structure du document, d'une manière très ćompacte, sous la forme d'un catalogue de données. Nous démontrons ceci de manière expérimentale dans la section 6.

Lecture sélective des entrées des jointures. L'efficacité de la valuation dépend à la fois de l'efficacité des entrées/sorties de données, et de l'efficacité de la jointure

2. Une jointure unique à plusieurs branches peut aussi être utilisée [BRU 02].

structurelle elle-même. Nous remarquons le fait suivant : appliquer une jointure sur deux ensembles d'identifiants peut réduire le coût d'accès à ces identifiants. En effet, la logique de contrôle de l'opérateur de jointure peut diriger la lecture des données de telle sorte que les entrées ne seront lues qu'en partie, sans affecter le résultat obtenu.

Une première classe de techniques proposées à cet effet se base sur les jointures en zig-zag [GAR 00], initialement proposées dans le domaine de la recherche d'information. Des travaux plus récents proposent de construire des index de type arbre B+ sur des identifiants structurels, afin d'évaluer des jointures structurelles en zig-zag [CHI 02]. Une deuxième classe de techniques à cet effet exploite les propriétés numériques des identifiants structurels, pour reconnaître les parties inutiles des entrées des jointures [TEU 03].

Un effet collatéral intéressant de notre schéma de partitionnement par chemins consiste à élargir l'usage des jointures en zig-zag à plus de situations que celles considérées dans [CHI 02]. En même temps, les jointures en zig-zag basées sur un stockage PC ne nécessitent pas de structure d'index spécialisées, telle que les arbres B+ utilisés en [CHI 02]. Les jointures zig-zag sont donc possibles sur notre stockage partitionné par les chemins. D'autres techniques de réduction des entrées des jointures, telles que celles décrites dans [TEU 03], sont conceptuellement orthogonales et peuvent aussi être appliquées.

Nous comparerons nos structures de stockages avec plusieurs modèles existants de stockage et indexation XML dans la section suivante, après avoir introduit dans les détails notre propre modèle.

3. Stockage à base de séquences de chemins

Nous décrivons ici les principes et la mise en œuvre du modèle de stockage basé sur les séquences de chemins. La section 3.1 décrit l'organisation logique des données, tandis que la section 3.2 discute des structures de données qui pourraient être utilisées pour mettre en œuvre ce modèle logique.

3.1. *Le modèle logique de partitionnement par chemin*

Notre modèle sépare la structure du document de son contenu, et stocke chaque ensemble d'objets de données similaires dans des séquences organisées dans l'ordre du document. Nous décomposons le document dans trois structures distinctes, que nous décrivons ici.

La première structure contient une représentation compacte de la structure de l'arbre XML. Nous attribuons des identifiants uniques et persistants à chaque élément d'un document XML, utilisant le schéma d'étiquettage [pre, post] employé dans [ALK 02, CHI 02, HAL 03]. Le numéro pre correspond à la position du début de cet élément dans le document, tandis que le numéro post correspond à la position de

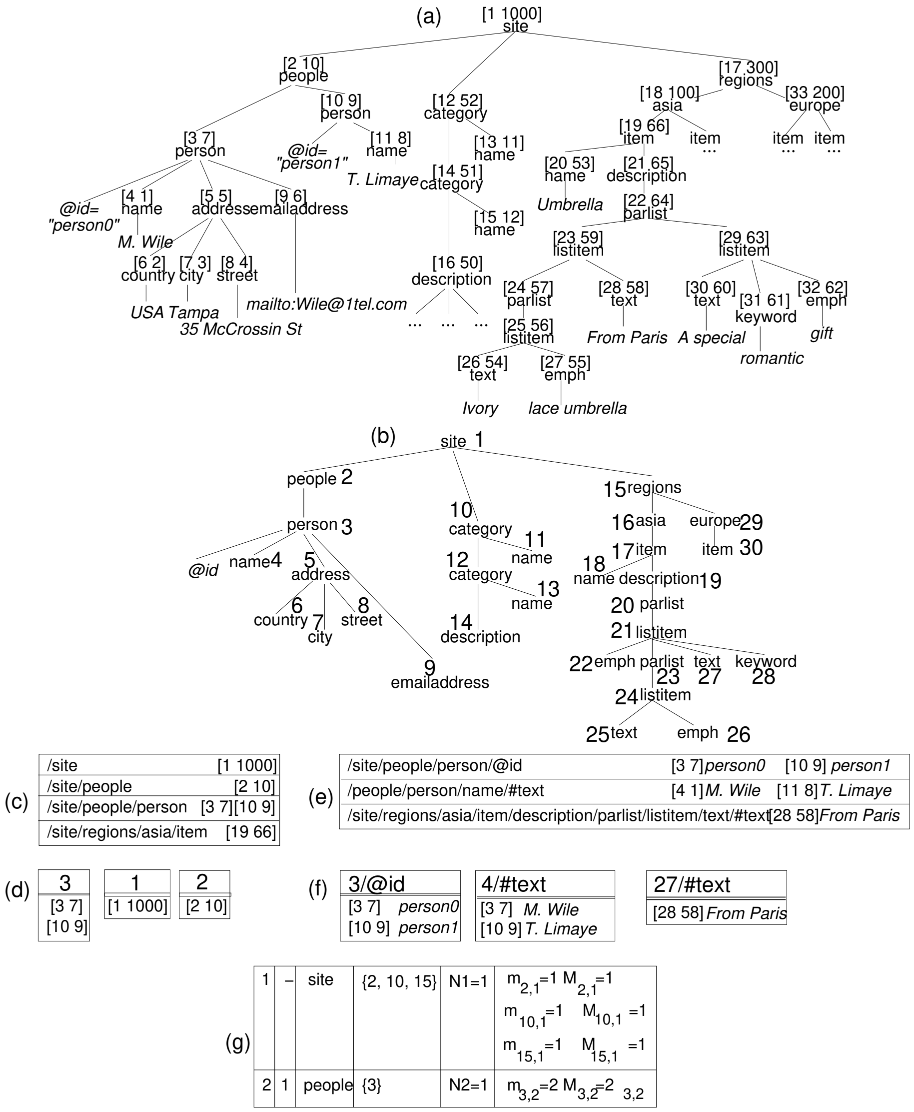


Figure 2. *Fragment d'un document XMark (a), son résumé structurel (b) et quelques-unes des structures de stockages qui en résultent (c-g)*

son tag de fin. En s'appuyant sur ce schéma, un élément e_1 est un ancêtre d'un autre élément e_2 si et seulement si $\mathsf{n}_1.\mathsf{pre} < \mathsf{n}_2.\mathsf{pre}$ et $\mathsf{n}_1.\mathsf{post} > \mathsf{n}_2.\mathsf{post}$. Par exemple, la figure 2(a) montre les identifiants de type [pre, post] de chaque élément juste au-dessus de cet élément.

Nous partitionnons ces identifiants selon le chemin de données sur lequel ils se trouvent. Logiquement, chaque partition est une séquence d'identifiants, dans l'ordre de leur numéro pre, qui refète l'ordre du document. Par exemple, la figure 2(c) présente quelques-unes des séquences qui résultent du document de la figure 2(a), pour les chemins site, /site/people, /site/regions/asia/item, etc.

Le schéma traditionnel d'étiquetage basé sur pre, post comprend aussi un attribut depth, reflétant la profondeur de chaque élément dans le document. Toutefois, le schéma de stockage partitionné par chemin rend superflu cet attribut, car il est stocké de manière implicite dans le chemin correspondant à chaque séquence.

La deuxième structure que nous utilisons stocke le contenu des éléments XML, et les valeurs de leurs attributs. Nous associons de telles valeurs à leur plus proche parent de type élément, plus précisément au numéro pre de cet identifiant, et nous stockons ces paires pre, valeur dans des séquences ordonnées dans l'ordre dicté par pre. Nous appellons une telle séquence un *container*, dans l'esprit du projet XMill [LIE 00]. Nous classifions les valeurs d'un container en entiers, réels et chaînes de caractères ; cette information est extraite des données mêmes. Par exemple, la figure 2(e) montre certains containers pour le document de la figure 2(a), pour les chemins /site/people/person/@id, /site/people/person/name/#txt, etc.

Les deux structures ci-dessus suffisent pour représenter un document XML, toutefois, nous y ajoutons une troisième structure qui joue le rôle d'un catalogue de données. Le résumé des chemins d'un document XML est un arbre dont les nœuds internes correspondent à des éléments XML, et dont les feuilles correspondent à des valeurs (textes ou attributs). Pour chaque chemin simple de la forme $l_1/l_2/\ldots/l_k$ correspondant à un ou plusieurs nœuds d'un document XML, il y a exactement un nœud atteint par le même chemin dans le résumé des chemins.

A chaque nœud x dans le résumé des chemins, nous attribuons un *numéro de chemin* distinct, de type entier. Ce numéro caractérise à la fois le nœud et le chemin depuis la racine du résumé structurel jusqu'au nœud x. Ainsi, chaque séquence d'identifiants correspondant à un certain chemin est associée à exactement un seul numéro de chemin. Chaque container est associé à une paire comprenant un numéro de chemin, et soit @attrName dans le cas d'un attribut, soit #text dans le cas d'un fragment textuel. La figure 2(b) représente le résumé des chemins pour le fragment XML montré à gauche. Les numéros des chemins apparaissent en gros caractères à côté des nœuds du résumé des chemins.

Un résumé des chemins correspond à un Dataguide [GOL 97] d'un document XML. Un Dataguide à été utilisé pour la première fois dans le systeme Lore, et par la suite, comme base pour l'indexation [MCH 99]. Le Dataguide d'un document XML correspond à un cas particulier de Template-Index [MIL 99] dans le cas des documents XML. Dans notre cas, le résumé des chemins fait partie du stockage lui-même ; en effet, il constitue le catalogue des données et il est nécessaire pour le traitement de requêtes, tandis qu'il était utilisé en tant qu'index optionnel dans [MCH 99]. De par son usage différent et son caractère obligatoire, nous préferons utiliser le terme *résumé structurel* pour en parler, plutôt que Dataguide.

Le résumé structurel comprend un ensemble de statistiques simples. Soit x un nœud de ce résumé, sur un chemin finissant par un élément nommé t, et soit y un enfant de x. Nous enregistrons :

– N_x : le nombre d'éléments trouvés sur le chemin x (la taille de la séquence d'identifiants correspondant à x) ;

– $m_{x,y}, M_{x,y}$: le nombre minimal, resp. maximal d'enfants y des éléments XML sur le chemin x.

Dataguides, enrichis avec les mêmes statistiques que celles présentées ci-dessus, ont été utilisés dans [ABO 01] pour estimer la cardinalité des requêtes de chemin. Les auteurs de [ABO 01] étudient également des méthodes d'élagage du Dataguide, pour s'assurer qu'il tient en mémoire. Dans XQueC, le résumé est nécessaire dans sa totalité pour le traitement de requêtes, ainsi il n'est pas possible de pratiquer un tel élagage. Nous montrons dans la section 6.3 que le résumé structurel est habituellement très petit en comparaison avec le document, et donc il peut être gardé en mémoire lors du traitement de requêtes.

Pour un document donné, nous utiliserons N pour désigner le nombre de nœuds dans le document, h pour sa hauteur et N_{PS} pour le nombre de nœuds dans son résumé structurel.

Un document XML peut être chargé dans un tel modèle de stockage en une seule passe, utilisant un parser XML à base d'évènements [SAX] qui produit des évènements au fur et à mesure qu'il traverse le document. La complexité de l'algorithme est linéaire en N et il utilise $O(h + N_{PS})$ unités de mémoire pour y construire le résumé structurel et y garder une pile nécessaire pour le chargement.

3.2. *Stockage physique pour le modèle logique partitionné sur les chemins*

Une propriété essentielle du modèle logique partitionné par les chemins est l'ordre dans les séquences d'identifiants et dans les containers. Ainsi, les structures de stockage physiques qui l'implémentent doivent de manière inhérente supporter l'ordre. Nous considérons deux structures ordonnées persistantes :

– arbres B+ : cette option est prise dans de nombreux travaux existants [BRU 02, CHI 02, HAL 03, ZHA 01]. Son avantage est sa robustesse et son bon support pour des mises à jour. Son désavantage (section 6.3) est sa taille rélativement importante, dûe à l'indexation intensive des identifiants.

– séquences persistantes : l'alternative que nous poursuivons ici est l'usage des séquences persistantes comme unité de stockage de base. L'avantage de ce modèle est son caractère très compact, ce qui réduit aussi les besoins en mémoire. Nous utilisons une implémentation complète de séquences persistantes, pourvue de fonctionnalités comme le verrouillage et le contrôle des transactions [BER]. Un système plus récent [LER 03] adopte un stockage organisé par séquences, qui est bien plus performant qu'une base de données relationelle classique dans des applications où l'ordre est critique (par exemple, séries temporelles). Son désavantage réside dans ses mauvaises performances au cas d'une mise à jour.

Notre travail adopte les séquences persistantes comme stockage physique.

La figure 2(d) représente le stockage physique des séquences d'identifiants de la figure 2(b). Les objets dans ces séquences ont une longueur constante, égale à la taille d'un identifiant. La figure 2(f) présente les séquences qui résultent des containers de la figure 2(c). Les entrées des containers ont des longueurs variables, dûes aux valeurs des données. Donc, nous utilisons des séquences de longueur variable.

Nous stockons le résumé structurel comme une séquence d'objets de longueur variable, comme illustré dans la figure 2(g). Il y a un objet pour chaque nœud, comprenant : le numéro du nœud, le numéro de son père, les numéros des enfants et les informations statistiques précedemment décrites. L'espace occupé par le résumé des chemins est linéaire en N_{PS}.

4. Plans de valuation sur un stockage partitionné par les chemins

Nous discutons maintenant la construction des plans de valuation des variables s'appuyant sur le modèle des séquences de chemin. Les plans de valuation produisent des identifiants des éléments correspondant aux variables.

Une expression *linéaire* de chemin (ou *elc*) est une expression du type $(v_i?)$ $(/|//)$ l_1 $(/|//)$ l_2 $\ldots(/|//)l_k$, où chaque l_i est soit une étiquette soit $*$, les l_is sont connectés par $/$ ou $//$, $k > 0$, et v_i est une variable optionnelle. Nous désignerons k comme étant la longueur d'une *elc*. Nous nommerons *elc simple* une *elc* du type $/l_1/l_2/\ldots/l_k$. En particulier, chaque nœud du résumé structure correspond à la *elc* simple qui le connecte à la racine. Nous disons que deux *elc simples* sont *reliées*, lorsque l'une est un préfixe de l'autre. Les nœuds du résumé structurel correspondant à une elc P sont ceux obtenus en évaluant P sur le résumé des chemins, considéré comme un arbre de données. Nous dirons aussi que les *elcs simples* de ces nœuds correspondent à P.

Nous considérons un modèle d'exécution générique, suivant l'interface Itérateur. Nous nommerons $op[i]$ la i-ème colonne de la sortie produite par op. Le type de chaque colonne peut être soit un type simple tel que chaîne de caractères, entier, etc, ou un identifiant structurel.

4.1. *Valuation d'une seule variable*

Le premier problème que nous considérons est d'associer à une seule variable une *elc*, telle que `$x` dans le fragment de requête `for $x in /site/people/person/name`.

Le partitionnement par chemin fournit un accès immédiat aux valuations pour cette variable : il suffit de lire la séquence d'identifiants qui y correspond, numérotée 4 dans la figure 2. Nous utilisons un itérateur IDScan(x), qui retourne les identifiants de la séquence de chemin associée à x ; IDScan remplit également l'attribut depth des identifiants d'éléments, avec la longueur de l'*elc* de x. Sa sortie est ordonnée et ne contient pas de doublons.

Considérons maintenant des *elc* plus complexes, comme par exemple dans le fragment for $x in //parlist//text. Nous identifions l'ensemble des nœuds du résumé structurel qui correspondent à //parlist//text, puis nous lisons toutes les séquences d'identifiants qui y correspondent, en les fusionnant en même temps. Les valuations ainsi obtenues respectent l'ordre du document, et ne comportent pas de doublon.

En général, nous pouvons récupérer les valuations pour $x en évaluant P sur le résumé structurel (ce qui produit un ensemble de *elc simples*), en lisant exactement les séquences d'identifiants qui sont utiles, et en les fusionnant. Aucune jointure n'est nécessaire.

Plus formellement, soit x_P le nombre de nœuds dans le résumé structurel qui correspondent à P, soit N_P le nombre de valuations pour P et soit b le facteur de remplissage d'un bloc mémoire. Les valuations pour P peuvent être obtenues avec un coût d'entrée/sortie de $O(N_P/b + x_P)$ et avec un coût CPU de $O(N_P * log(x_P))$. Les besoins en mémoire de cet algorithme sont de l'ordre de $O(N_{PS} + x_P)$.

Le coût d'entrée/sortie est dû exactement à la lecture des identifiants qui font partie de la valuation. Chaque séquence d'identifiants est physiquement contigue, mais toutes les séquences correspondant à x_P peuvent ne pas être contigues entre elles, d'où les x_P lectures supplémentaires. Le coût CPU correspond à la fusion des x_P séquences d'identifiants ordonnées ; l'opérateur Merge utilise une queue de priorités. L'occupation mémoire reflète le besoin de tenir en mémoire le résumé structurel, ainsi que la mémoire nécessaire à la gestion de la queue.

Afin d'évaluer une *elc* P sur le résumé structurel, nous le traversons du haut en bas, identifiant les étiquettes de P de gauche à droite, et en ajoutant les nœuds correspondant à la dernière étiquette de P à l'ensemble des résultats. La complexité en opérations de cette évaluation est en $O(N_{PS})$.

Par exemple, pour trouver des valuations pour //asia//item, nous commencons par le nœud 1 avec P, et nous cherchons des correspondances pour sa première étiquette asia. Puisque la racine a une étiquette différente, nous propageons la recherche de P vers tous les fils de la racine, ce qui le propage davantage. La recherche propagée sur la branche regions/asia produira une correspondance pour asia, et la recherche pour item sera propagée vers les enfants, etc.

Valuations avec des prédicats simples : considérons des sélections simples sur des valeurs textuelles, ou des attributs de $x, comme dans :

for $x in //people//person[@id="person0"]

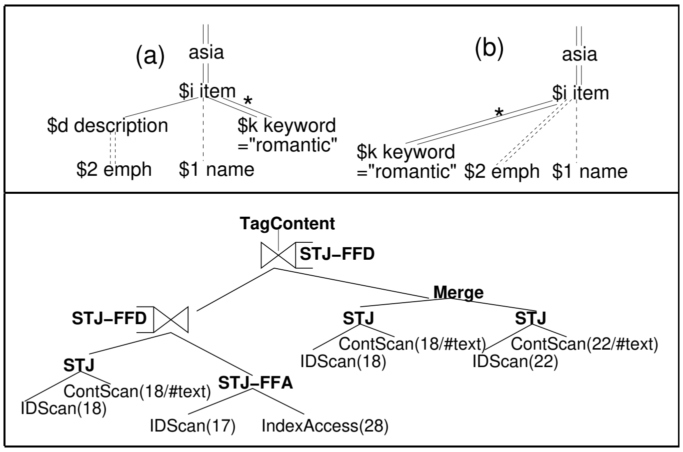


Figure 3. *Motif d'arbre généralisé (a) et motif réduit (b) pour la requête de l'exemple 1 ; plan d'exécution*

Dans de tels cas, nous devons accéder au container avec person/@id. Pour cela, nous utilisons un opérateur ContScan(x), où x est un chemin de container comme 3/@id dans la figure 2, qui retourne la séquence ordonnée des tuples (pre, value). Sur le ContScan, une sélection peut être appliquée. S'il existe un index sur le container, nous utiliserons l'opérateur IdxAccess pour accéder seulement aux tuples désirés. De plus, nous appliquons une jointure structurelle avec IDScan(4), afin de récupérer des identifiants de la forme [pre,post,depth].

4.2. *Valuations de plusieurs variables reliées*

Valuation des variables avec des prédicats complexes. Ce cas peut être vu comme une valuation de plusieurs variables reliées, sur certaines desquelles il y a des prédicats simples. Nous traitons ce cas dans la suite.

Comme exemple d'une valuation de plusieurs variables, la figure 3(a) montre les variables de la requête de l'exemple 1, structurées dans un motif d'arbre généralisé [CHE 03]. Nous utilisons des lignes simples pour des relations /, et des lignes doubles pour //. Les lignes discontinues représentent des jointures externes-gauche, considérant le nœud parent comme étant à gauche. Les arêtes marquées d'un $*$ correspondent à des relations de semi-jointure à gauche, en considérant aussi le nœud parent à gauche. Nous avons attribué des noms ad-hoc 1 et 2 aux nœuds correspondant aux expressions de la clause return.

La valuation de telles variables procède en deux étapes. Dans un premier temps, nous déterminons le nombre minimal de jeux de données auxquelles il faut accéder pour trouver des valuations des variables, en analysant les relations entre ces variables et le résumé structurel. Lors de cette étape, nous pourrions aussi découvrir des variables inutiles ou des nouvelles relations entre ces variables. Dans un deuxième temps, nous construisons des plans de valuations, en nous appuyant sur les connaissances obtenues dans la première étape.

Inférence des chemins des variables. Nous commencons par calculer, pour chaque variable, l'ensemble des chemins dans le résumé des chemins qui pourrait fournir des valuations pour cette variable. Dans cette section, nous utiliserons une paire de variables u, v d'une requête, telles que u est un parent de v dans le motif de la requête ; nous désignerons leurs ensembles de chemins respectifs par s_u et s_v.

Nous calculons ces ensembles en traversant parallèlement le motif de la requête et le résumé structurel, utilisant s_u comme point de départ pour déterminer s_v. Par exemple, pour la requête dans la figure 3 et le résumé structurel de la figure 2, s_i est /site/regions/asia/item. A partir du nœud de s_i, nous cherchons des correspondances pour /description, et s_d est /site/regions/asia/item/description. De même, à partir de s_i nous cherchons les correspondances pour //keyword, ce qui produit l'ensemble s_k=28 (nous utilisons les numéros des chemins donnés dans la figure 2(b)). A partir de s_d et s_i, nous obtenons s_1=18 pour $1 et s_2=22,26 pour $2.

Transformations des semi-jointures. Pendant l'inférence de chemins, pour chaque arête connectant une variable v à une autre variable enfant u, et pour chaque paire de chemins reliés $x \in s_u$, $y \in s_v$, nous calculons le *facteur de chemin maximal* $Mpf_{x,y}$. Ce facteur est le plus grand nombre M_{x_i,y_i} rencontré en traversant le résumé des chemins depuis le nœud x jusqu'au nœud y. Par exemple, pour l'arête qui connecte $1 à $2 dans la figure 3(a), nous obtenons $Mpf_{17,22}$ et $Mpf_{17,26}$; $Mpf_{17,22}$ est le plus grand parmi $M_{17,19}$, $M_{19,20}$, $M_{20,21}$, et $M_{21,22}$.

Si $Mpf_{x,y} \leq 1$ et si l'arête qui unit u à v est une semi-jointure, nous la transformons dans une arête de jointure, puisque la jointure et la semi-jointure coïncident. Ceci pourrait rendre plus efficace la valuation des variables de la clause for.

Par exemple, si $Mpf_{16,28} \leq 1$ dans la figure 2, l'exécution de la requête de l'exemple 1 pourrait commencer par un accès à $k, suivi immédiatement par une jointure avec les valuations pour $i. Sans cette optimisation, une étape d'élimination de doublons sur $i aurait été nécessaire lors de la combinaison des valuations.

Elimination des branches de jointure. De même, nous calculons le *facteur de chemin minimal* $mpf_{x,y}$ pour chaque paire de chemins reliés x et y, comme étant la valeur m_{x_i,y_i} la plus petite qui se trouve le long du chemin de x à y. Si l'arête entre u est v est une arête de jointure, $mpf_{x,y} = 1$ et $Mpf_{x,y} = 1$ pour tous les chemins reliés x, y, et il n'y a pas de prédicat directement sur v, alors nous pouvons éliminer v de la requête et coller les descendants de v directement à u. Par exemple, sur la requête de la figure 3(a), si tous les objets d'Asie ont exactement une description, la variable $d est inutile et elle est éliminée, ce qui conduit à la requête réduite de la figure 3(b).

Elagage des chemins des variables. Nous utilisons par la suite chaque ensemble s_v pour élaguer des chemins inutiles de s_u, où u est une variable parente de v dans la requête.

Par exemple, considérons la requête :

for $p in //parlist, $k in $p//text return $k

L'inférence de chemins dans ce cas produit s_p={20, 23}, et s_k={26}.

Puisque nous savons par le résumé des chemins que 26 est un descendant de 20 et non pas de 23, nous éliminons le chemin inutile 23 de s_p, qui est donc réduit à 20.

Cet élagage ne s'applique que lorsque dans le GTP, x et y sont connectés par des arêtes continues seulement. Il ne s'applique pas à des enfants connectés à leurs parents par des arêtes discontinues, puisque la présence de tels enfants n'est pas requise par la requête, pour garder une valeur du parent : même des parents (ou ancêtres) n'ayant pas ces descendants optionnels doivent produire des résultats.

Ce même élagage peut trouver que la réponse à une requête est vide, dû à un ensemble vide de chemins pour une variable connectée à l'aide de jointures et semi-jointures uniquement à la racine.

Enrichissement de requêtes. Enfin, des nouvelles relations structurelles entre les variables peuvent être découvertes. Par exemple, considérons le fragment :

for $i in //item, $p in $i//parlist, $t in $i//text

Après l'inférence des chemins et l'élagage, tous les chemins de $t sont des descendants d'un certain chemin de $p. Ceci se traduit par une nouvelle arête de jointure entre $p et $t, ce qui fournit de nouvelles possibilites de combinaison des jointures (sans rajouter ni enlever des nœuds de la requête).

Algorithme d'inférence de chemins. L'algorithme est décrit dans la figure 4.2. Il comprend une fonction de traversée récursive traverse qui est appelée sur un nœud pv du résumé des chemins, étant donnés :

– son parent dans le résumé des chemins, pu ;

– le dernier nœud u de la requête pour lequel une correspondance à été trouvée pour un ancêtre de pv ;

– le dernier nœud x de la requête qui correspond à une variable, et pour lequel une correspondance a été trouvée dans un ancêtre de pv[3] ;

– le chemin px auquel ce nœud x avait correspondu.

L'appel à traverse sur pv a pour but de mettre en correspondance pv avec l'un des enfants de u dans le pattern de la requête. Pour chaque tel nœud enfant v_i, nous vérifions si l'étiquette de pv correspond à celle de v_i. Si, de plus, v_i est une variable, alors nous marquons le chemin que nous avons trouvé pour lui, étant donné que sa variable

3. Tous les nœuds du motif de la requête ne correspondent pas à des variables ; par exemple, le nœud asia dans la figure 3 n'est pas une variable, toutefois il faut lui trouver des correspondances avant de le faire pour ses descendants.

parente x a été mise en correspondance avec le chemin p_x (ligne 5 de l'algorithme). En même temps, nous mettons à jour $mpf_{px,pv}$ et $Mpf_{px,pv}$ avec les statistiques m et M stockées pour pu et pv (ligne 6).

Ensuite, il faut propager les correspondances plus loin, essayant de mettre en correspondance les enfants de v_i (dans la requête) avec des enfants de p_v (dans le résumé des chemins), par des appels recursifs (lignes 8-9). Pour cet appel, la dernière variable pour laquelle une correspondance a été trouvée est x (qui a pu changer lors de la correspondance de v_i), et le dernier nœud mis en correspondance est maintenant v_i.

Enfin, si l'arête entre u et v_i est une arête ancêtre-descendant, il faut aussi traverser les enfants de pv avec les mêmes paramètres que pour pv. Ceci garantit que les arêtes de la requête marquées // sont correctement mises en correspondance à toute profondeur dans le résumé des chemins.

	Algorithm traverse(GTPNodes u,x, PSNodes pu, px, pv)
1	foreach GTP node v_i child of u
2	$nextX$= x ; $nextPX$= px
3	if (the tag of pv matches the tag of v_i)
4	if (v_i is a variable node)
5	mark (x, v_i, px, pv)
6	upd ($mpf_{px,pv}$, $Mpf_{px,pv}$, $m_{pu,pv}$, $M_{pu,pv}$)
7	$nextX$= v_i ; $nextPX$= pv
8	foreach PSNode pw child of pv
9	traverse (v_i, $nextX$, pv, $nextPX$, pw)
10	if (the edge u-v_i is ancestor-descendent)
11	foreach PSNode pw child of pv
12	traverse(u, x, pv, px, pw)

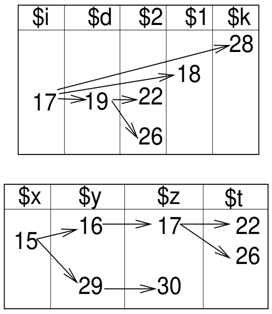


Figure 4. *L'algorithme d'inférence de chemins et quelques exemples de résultats*

Le processus d'inférence produit les listes de chemins trouvées pour chaque variable, groupées hiérarchiquement selon les variables de la requête. Dans la figure 4.2, en haut à droite, nous avons représenté le résultat pour la requête de l'exemple 1. Chaque chemin trouvé pour une variable pointe vers les chemins correspondants trouvés pour ses enfants dans la requête. (Les flèches conservent la sémantique des arêtes de la requête ; les flèches partant d'un même chemin d'une variable et atteignant des chemins différents d'une autre variable, ont une sémantique disjonctive.) Les flèches sont décorées avec les facteurs mpf et Mpf calculés lors de l'inférence des chemins.

Ce résultat permet déjà d'éliminer les chemins auxquels il manque des descendants obligatoires. Par exemple, dans la figure 4.2 en bas à droite, nous montrons les chemins résultants pour les variables de :

$x in //regions, $y in $x/*,
$z in $y/item, $t in $z//emph

sur le résumé des chemins de la figure 2. Puisque le chemin 30 pour $z n'a pas de chemin descendant pour $t, nous éliminons 30, et par la suite aussi 29.

La transformation des semi-jointures et l'élimination des chemins s'applique aussi sur (et peuvent réduire) le résultat de l'élagage de chemins. Enfin, pour déduire des nouvelles arêtes dans la requête, nous analysons les chemins ayant survécu aux étapes précédentes. Ce processus peut être rajouté dans l'exécution de la fonction traverse, utilisant une matrice de N_{PS}^2 bits pour marquer quel chemin est un descendant de quel autre.

Construction de plans de valuation. Utilisant les ensembles de chemins produit par les étapes ci-dessus, nous construisons des plans de valuations individuels, comme nous l'avons décrit dans la section 4.1. Pour la requête de l'exemple 1, ceci produit :

$i	IDScan(17)
$k	Join(IDScan(28), Filter(ContScan(28/#text))
$1	IDScan(19)
$2	Merge(IDScan(22), IDScan(26))

Nous remarquons que $d a été éliminée. Ces plans sont très sélectifs, puisqu'ils n'accèdent qu'aux identifiants dont les chemins leur permettent de contribuer aux résultats de la requête.

Métadonnées et statistiques pour les plans de valuation. Soit $|op|$ l'estimation de la cardinalité de l'opérateur *op*. Si $op[i]$ est de type ID, nous appelons $cList(op, i)$ la liste des numéros de chemin sur lesquelles les identifiants de $op[i]$ peuvent se trouver.

Pour des plans IDScan(x), $cList(op)$={x}, $|op|{=}N_x$. Pour des plans du type Merge(IDScan(x_1),. . ., IDScan(x_k)), $pList(op)$={x_1,x_2,. . .x_k}, nous estimons $|op|$ comme étant $\sum_{x \in pList(op)} N_x$. Ces estimations ont été introduites pour la première fois dans [ABO 01].

Pour des plans de type Filter ou IDAccess, une estimation de la sélectivité des prédicats est nécessaire pour estimer $|op|$. Des statistiques sur des containers peuvent être utilisées à cet effet.

5. Combinaison des plans de valuation

Les plans de valuation sont combinés progressivement par des jointures et des jointures structurelles, comme décrit dans [WU 03, CHE 03]. Nous nous concentrerons sur des techniques efficaces qui deviennent possibles grâce au modèle de stockage basé sur les séquences de chemin, et qui peuvent être intégrées de manière profitable dans l'étape de combinaison.

Dans la section 5.1, nous présentons une condition suffisante pour garantir de bonnes propriétés d'ordre dans les jointures structurelles. La section 5.2 décrit des algorithmes de jointure plus complexes, rendus possibles par notre modèle de stockage partitionné par les chemins. Nous discutons ensuite brièvement la construction des plans de requête.

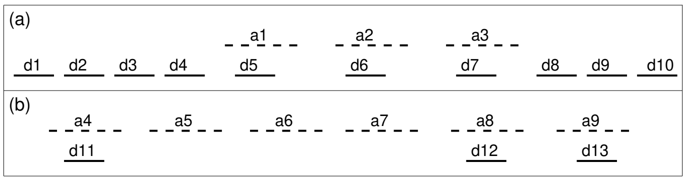


Figure 5. *Jointures structurelles où des parties de l'entrée peuvent ne pas être lues*

5.1. *Algorithmes de jointure préservant l'ordre*

Les algorithmes *StackTreeDesc (STD)* et *StackTreeAnc (STA)* [ALK 02] s'appliquent directement dans notre contexte. Ils calculent la jointure de deux relations, selon des relations structurelles satisfaites par les identifiants de type [pre, post, depth] que ces relations contiennent.

STA et STD recoivent en entrée des identifiants dans l'ordre du document ; STA préserve l'ordre des identifiants des ancêtres, et STD l'ordre des identifiants des descendants. Les deux algorithmes utilisent une pile ; STA utilise de plus d'autres structures de données. Les deux algorithmes fonctionnent en *pipeline*.

Pour éviter les opérations de tri, il serait intéressant d'avoir un opérateur de jointure structurelle qui préserve à la fois l'ordre des identifiants des ancêtres et des descedents ; toutefois, ce n'est pas toujours possible. Une jointure structurelle préserve les deux ordres des entrées si et seulement si les identifiants dans l'ancêtre ne contiennent pas de pairs d'identifiants ancêtre-descendant [ALK 02, CHE 03]. Cette condition dépend des données, ainsi sa vérification est très coûteuse.

Nous fournissons une condition suffisante, qui peut être établie plus facilement :

Soient op_1 et op_2 deux opérateurs, tels que $op_1[i]$ et $op_2[j]$ contiennent des identifiants d'éléments. Si les chemins dans $cList(op_1, i)$ sont tous mutuellement non-reliés, alors une jointure structurelle combinant des identifiants d'ancêtres dans $op_1[i]$ avec des identifiants de descendants dans $op_2[j]$ préserve les deux ordres des identifiants dans l'entrée.

Etant donné un ensemble de chemins $cList$ de taille k, cette condition peut être vérifiée dans un temps $O(k * h)$. Pour chaque nœud dans $cList$, nous naviguons vers la racine ; si nous rencontrons un autre nœud de $cList$, alors les chemins ne sont pas mutuellement non reliés.

5.2. *Les algorithmes STD-FastForward*

Une jointure structurelle peut éviter de lire des parties de ses entrées. La figure 5 en fournit deux exemples. Les identifiants des éléments sont montrés comme des segments [pre,post] discontinus pour les IDs de l'opérateur anc, et continus pour ceux de lópérateurs $desc$. Un identifiant a de anc est un ancêtre d'un identifiant d de $desc$, si et seulement si le segment de d se trouve en dessous le segment de a.

Dans la figure 5(a), le premier descendant de a_1 est d_5. Donc, la lecture de $d_1 - d_4$ peut être évitée, car ils ne produiront pas de résultats de jointure. En échange, la jointure devrait avancer $desc$ jusqu'au premier d qui pourrait être un descendant de a_1, c'est-à-dire, tel que d.pre$>a_1$.pre. Puisque les tuples $desc$ sont triés sur pre, il est possible d'avancer $desc$ de la sorte.

Dans la figure 5(b), après la paire a_4, d_{11}, la jointure pourrait éviter $a_5 - a_7$ et avancer anc au premier a qui pourrait être un ancêtre de d_{12}, c'est-à-dire tel que a.post$>d_{12}$.post. Ceci nécessite de sauter sur l'attribut post, mais les entrées sont triées sur pre ; ce saut n'est possible que si les deux ordres coïncident. De par la sémantique de pre et post, il est facile de voir que ces ordres coïncident dans un ensemble d'identifiants si et seulement si cet ensemble ne contient pas de paire ancêtre-descendant.

Une condition suffisante s'applique ici. Etant donné un opérateur *op* tel que *op*$[i]$ est de type ID, si $cList(op, i)$ sont mutuellement non reliées, alors les ordres pre et post dans *op*$[i]$ coïncident.

Nous proposons l'opérateur de jointure structurelle *STJ-FastForward* (STJ-FF) ; il dérive de l'algorithme StackTreeDesc. STJ-FF peut éviter des parties de l'entrée à la fois en se basant sur anc.post et sur $desc$.pre. Il utilise deux nouvelles méthodes fournies par anc et $desc$:

– *nextFF-pre(col, int n)* positionne un opérateur *op* sur le premier tuple de sortie ayant *op*$[col]$.pre$\geq n$. Si un tel tuple n'existe pas, nextFF-pre retourne faux ;

– *nextFF-post(col, int n)* positionne un opérateur *op* sur le premier tuple de sortie ayant *op*$[col]$.post$\geq n$. Si un tel tuple n'existe pas, nextFF-pre retourne faux.

Utilisant ces méthodes, un opérateur parent commande à ses enfants de produire uniquement des tuples ayant certaines valeurs. Ces méthodes peuvent être appellées uniquement sur les opérateurs *op* dont la sortie est triée par *op*$[col]$.pre ; de plus, la méthode nextFF-post exige que les chemins dans $pList(op, col)$ soient mutuellement non reliés.

Par exemple, soit *op*=STJ(IDScan(17), Merge(IDScan(21), IDScan(24))) avec les notations des chemins de la figure 2. *op* permet : nextFF-pre sur les colonnes 0 et 1, et nextFF-post sur la colonne 0 seulement, puisque 21 et 24 sont reliées.

Dans la figure 5(a), après le saut initial à d_5, la région $d_5 - d_7$ contient des identifiants utiles. Ainsi, après avoir trouvé d_5, une recherche binaire pour trouver d_6 ferait un effort inutile. Par conséquent, nous implémentons les opérateurs qui sautent de manière optimiste dans les opérateurs IDScan et ContScan : la prochaine valeur disponible est examinée, et seulement si elle ne satisfait pas la condition de saut, nous appliquons une recherche binaire.

Les sauts dans des jointures structurelles peuvent détériorer l'efficacité du cache. Pour qu'elle soit profitable, cette technique doit être utilisée seulement au-dessus d'une certaine fréquence des identifiants inutiles. Si un bloc de données est lu pour chaque ID, éviter les comparaisons pour les autres ne va pas apporter de bénefices visibles. Nous démontrons ceci de manière expérimentale dans la section 6.5.

Les situations de la figure 5 ne se produisent pas toujours simultanément. Par exemple, considérons la requête : for $x in //person[id="person0"], $y in $x/name. Lorsque les valuations des variables $x et $y sont combinées, ceci produira de nombreuses valuations inutiles pour $y et une seule valuation utile pour x. En conséquence, nous spécialisons l'opérateur STJ-FF dans deux variantes : STJ-FFA, qui ne saute que des tuples de l'ancêtre, et STJ-FFD, qui ne saute que des tuples du descendant, et fait avancer anc par des appels next(). Nous démontrons leur intérêt dans la section 6.5.

Optimisation. Pour évaluer une certaine jointure structurelle, l'optimiseur examine les listes de chemin des operandes. Chaque fois que les listes $cList$ des ancêtres ne sont pas reliées, il sait que les deux ordres seront respectés. Par exemple, toutes les jointures structurelles dérivées des requêtes XMark ne présentent que des chemins anc mutuellement non-reliés ! Ainsi, STJ-FFD peut toujours être utilisé, dès qu'il est un bon candidat.

L'optimiseur doit aussi savoir quand utiliser des jointures structurelles qui évitent des parties des entrées, telles que celles décrites dans la section 5.2. Pour cela, il utilise des estimations de cardinalité sur les plans de valuation et sur les jointures structurelles. Par exemple, pour la requête for $x in //person[@id="person1"], $y in $x//age, STJ-FFA est un bon candidat de jointure, puisque $x correspond à un seul élément person.

6. Validation expérimentale

Dans cette section, nous évaluons les performances du modèle de stockage proposé. Nous présentons d'abord l'environnement matériel et logiciel dans la section 6.1. Nous présentons la base de la comparaison dans la section 6.2, et discutons la taille occupée par le stockage dans la section 6.3. Nous étudions la performance des plans de valuation pour le cas d'une seule variable (section 6.4) et pour le cas de plusieurs variables (section 6.5). Nous présentons nos conclusions dans la section 6.6.

6.1. *Environnement matériel et logiciel*

Nous avons implémenté le modèle PC dans le système GeX, un prototype développé en Java à partir de nos travaux précédents sur le système XQueC [ARI 04]. Le système comporte environ 170 classes ou 60000 lignes de code. Le stockage persistant est construit au-dessus de la librairie BerkeleyDB [BER]. Nous utilisons un ordinateur portable D800, équipé d'un processeur Pentium IVM à 1.4 GHz et 1Gb de RAM, sous RedHat Linux 9.0. Tous les temps d'exécution sont pris en moyenne sur 20 exécutions

Document	Taille (Mb)
XMark	118
DBLP	133
SwissProt	114

Tableau 1. *Documents utilisés pour nos mesures*

à chaud. Les jeux de données utilisés dans ces mesures et leurs tailles sont décrits dans le tableau 1.

6.2. *Base de comparaison : PE et OPE*

Nous étudions les performances du modèle de stockage à base de séquences (PC) en le comparant avec son concurrent le plus proche, à savoir le partitionnement par étiquette (PE). Nous décrivons les principes de ce modèle et les situations où le partitionnement par chemin fonctionne mieux dans la section 2.

Dans le modèle PE, les documents XML sont stockés comme des arbres persistants ou des tables relationnelles. De plus, des index sous la forme d'arbres B+ sont ajoutés à ce stockage, indexant tous les identifiants par la clé (tag+pre). Le traitement de requêtes basé sur les index est la meilleure stratégie. Nous comparons donc notre technique de traitement de requêtes avec les techniques décrites dans [ALK 02, WU 03, CHE 03] basées sur les index par étiquette. Dans ce but, nous avons mis en œuvre aussi PE, dans deux variantes : l'une basée sur des arbres B+, et l'autre basée sur des séquences partitionnées par l'étiquette. Cette dernière peut être vue comme la fusion de toutes les séquences (ou containers) qui finissent sur une même étiquette, dans une séquence unique.

PE optimisé : pour diminuer le handicap de performance de PE, nous comparons les performances non seulement entre PE et PC, mais aussi avec PEO (partitionnement par étiquette optimisé) que nous avons décrit dans la section 2. Intuitivement, PEO construit des valuations sur un stockage partitionné par l'étiquette, mais les opérations à appliquer afin de trouver des valuations pour les variables ont été simplifiées autant que possible, en s'appuyant sur le résumé des chemins. Ainsi, la performance de PEO est la meilleure qui puisse être atteinte en utilisant un stockage partitionné par chemins, en supposant que l'optimiseur applique toutes les simplifications possibles, et qu'il connaisse également le meilleur ordonnancement de jointures.

6.3. *Taille du stockage*

La figure 6 compare la taille du stockage pour PC et PE, utilisant des arbres B+ comme dans [CHE 03, HAL 03, ZHA 01]. Pour XMark, la structure du document stockée par PC occupe environ 10 % de la taille du document. Utilisant PE, la taille est

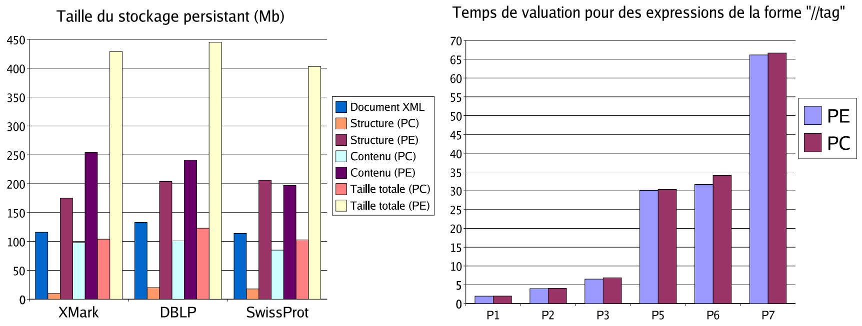


Figure 6. *Comparaison de la taille du stockage pour PE et PC (gauche), surcoût de la fusion de séquence lors de la valuation d'une variable (droite)*

20 fois plus grande. Ceci est dû à la présence des étiquettes dans les clés de l'arbre B+ (tandis que PC ne stocke les étiquettes que dans le résumé des chemins) et aux pages d'index intermédiaires. Globalement, la taille du document est multipliée par 3.5-4 lorsqu'il est stocké avec PE, un facteur proche du facteur 4 décrit dans [CHE 03] ; utilisant PC, la taille totale du stockage est légèrement plus petite que le document.

Le résumé des chemins occupe 26 Kb pour XMark, 6 Kb pour DBLP, 10 Kb pour SwissProt ; N_{PS} est respectivement de 514, 122 et 117. Ceci valide la supposition qu'il peut être gardé en mémoire lors du traitement de requêtes.

6.4. *Valuations pour une variable*

Nous commençons par considérer la valuation d'une seule variable correspondant à une *elc*. Considérons des *elc* dont les étiquettes sont connectées par des //.

Le cas le plus favorable pour PE est une variable à laquelle correspond une *elc* de la forme //tag : PC doit fusionner plusieurs opérateurs IDScan (section 4.1), tandis que PE n'a qu'à récupérer les identifiants des éléments d'une étiquette donnée. Lorsqu'un seul nœud du résumé des chemins correspond à //tag, PE et PC coïncident.

La figure 6 compare les temps d'exécution des plans de valuations lorsque PE et PC sont différents ; les requêtes sont décrites dans le tableau 2.

Les cardinalités des plans de valuations varient de 2000 (P_1) à 720000 (P_7). Le surcoût de PC, dû à la lecture et à la fusion de différentes séquences, est négligeable. Plusieurs facteurs expliquent ce fait : (i) l'opérateur de fusion est en *pipeline* et a un coût réduit grâce à l'arbre des entrées qu'il construit ; (ii) les identifiants sont stockés de manière très compacte par PC et sont lus en séquence, donc il n'y a pas de mauvais usage du cache ; (iii) le facteur de fusion est de l'ordre des centaines, ce qui permet de garder en mémoire un bloc pour chaque séquence d'entrées.

Code de la requête	Requête	Document (voir tableau 1)
P_1	//item	XMark
P_2	//description	XMark
P_3	//bold	XMark
P_4	//category	XMark
P_5	//title	DBLP
P_6	//author	DBLP
P_7	//Descr	SwissProt

Tableau 2. *Requêtes linéaires utilisées dans les mesures de la figure 6 (à droite)*

Valuation des variables à des chemins plus longs : dans de telles situation, PE est handicapé par son accès aux données trop peu sélectif. Par exemple, P /site/people/person/name. Dans ce cas, PE permet seulement d'accéder à tous les identifiants de name. La plupart de ces identifiants ne sont pas sur le bon chemin : ils peuvent correspondre à des éléments sur les chemins 11, 18 (voir figure 2), ce qui conduit à des nombreuses lectures de données inutiles. Pour séparer seulement les noms des personnes, nous devons accéder à tous les identifiants des éléments person (donc lire davantage de données !) et appliquer une jointure structurelle entre les deux. De plus, nous devons lire tous les identifiants des éléments site et people, et appliquer deux jointures successives, pour séparer juste les bons noms. En général, pour construire des plans de valuation de variables à une *elc* de longueur k, PE doit lire tous les identifiants des éléments impliqués et calculer une jointure à k voies, tandis que PC n'a besoin que de lire le résultat.

Pour la requête for $x in //parlist//text, TP doit calculer les jointures de tous les identifiants d'élément parlist avec tous les identifiants d'éléments text ; mais ceci pose un nouveau problème. Un élément text peut être un descendant de deux ou plusieurs éléments parlist, comme c'est le cas de l'élément [26,54] dans la figure 2. De tels éléments text apparaîtront deux fois dans le résultat de la jointure. Ainsi, une élimination de doublons sera nécessaire.

La figure 7 (gauche) considère des *elc* de longueur 2, qui sont décrites dans le tableau 3. De P_8 à P_9, PE et PC coïncident ; PE fait une jointure de plus ainsi qu'une élimination de doublons. De P_{10} à P_{17}, OPE n'économise que sur l'élimination des doublons, qui est simple puisque l'ordre du résultat de la jointure était favorable. La différence de performance entre PC et PEO reflète la proportion des identifiants inutilement lus par PEO dû au modèle de stockage partitionné par l'étiquette. Cette proportion atteint 430 pour P_{14} ; PC est très rapide pour P_{14}, P_{15} et P_{17}.

Dans la figure 7 (à droite) nous mesurons des *elc* de longueur 3, qui sont aussi décrites dans le tableau 3. La performance de PE et PEO est déterminée par l'effort de lecture des données ; PC peut être des centaines de fois plus performant.

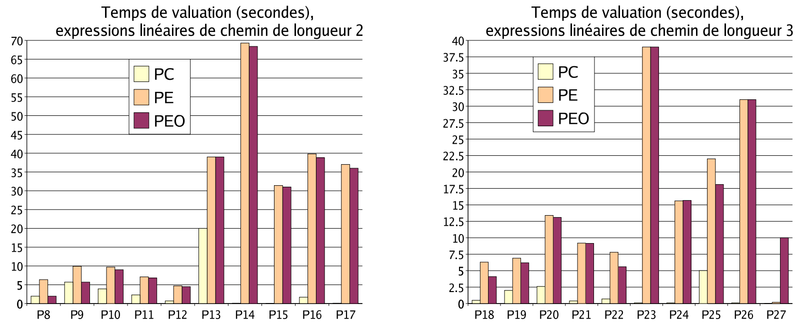


Figure 7. *Valuations pour des expressions linéaires de chemin de longueur 2 et 3*

De ces mesures, nous tirons la conclusion que PC est systématiquement plus efficace que PE, aussi bien que PEO, pour la tâche de valuation des variables. De plus, le handicap de PE grandit avec la longueur de l'expression de chemin considérée.

6.5. *Combinaison des plans de valuation*

La figure 8 compare les performances de PE et PC pour la combinaison de deux, respectivement trois variables reliées. Dans ces cas, PE ne peut pas profiter d'une quelconque optimisation, puisque les lectures des données et les jointures structurelles sont strictement nécessaires afin d'obtenir des paires de valuations. De plus, PC doit aussi lire plusieurs entrées et en calculer la jointure. Dans la figure 8, PC et PE utilisent encore une fois le meilleur ordonnancement de jointures disponible.

A gauche dans la figure 8, nous considérons à nouveau les chemins P_8 à P_{17} de la figure 7, mais cette fois-ci nous remplacons `for $x in //tag`$_1$`//tag`$_2$ par `for $x in //tag`$_1$, `$y in $x//tag`$_2$. Par exemple, dans la figure 8 à gauche, P'_8 représente le motif à deux variables `for $x in //description, $y in $x//parlist`. Pour P'_8 et P'_9, PE et PC sont très proches, puisque PE ne fait pas de lecture inutile (tous les éléments `parlist` se trouvent sous des éléments `description`, et de même pour P'_9), et le coût est dominé par l'accès aux données. Pour P'_{10}-P'_{17}, l'accès peu sélectif de PE se retrouve directement dans une différence de performance très importante avec PC ; les courbes suivent de près celles de la figure 7 à gauche.

A droite dans la figure 8, nous considérons des motifs à trois variables, représentés juste en-dessous du graphe respectif. Même très petits, ces motifs illustrent bien les problèmes de performance impliqués. Nous avons choisi des *elc* d'un seul tag seulement, pour avoir le meilleur cas possible pour PE ; si des chemins plus longs connectent les variables, PE fera des lectures et des jointures inutiles, comme expliqué précédemment.

Code	Requête	Jeu de données (voir tableau 1)
P_8	//description//parlist	XMark
P_9	//parlist//listitem	XMark
P_{10}	//item//keyword	XMark
P_{11}	//person//name	XMark
P_{12}	//category//name	XMark
P_{13}	//article//title	DBLP
P_{14}	//book//author	DBLP
P_{15}	//book//title	DBLP
P_{16}	//METAL/Descr	SwissProt
P_{17}	//LIPID//Descr	SwissProt
P_{18}	//europe//item//description	XMark
P_{19}	//regions//item//description	XMark
P_{20}	//item//description//keyword	XMark
P_{21}	//europe//parlist//bold	XMark
P_{22}	//europe//parlist//listitem	XMark
P_{23}	//Entry//METAL//Descr	SwissProt
P_{24}	//categories//listitem//text	XMark
P_{25}	//parlist//listitem//text	XMark
P_{26}	//dblp//book//author	DBLP
P_{27}	//dblp//book//title	DBLP

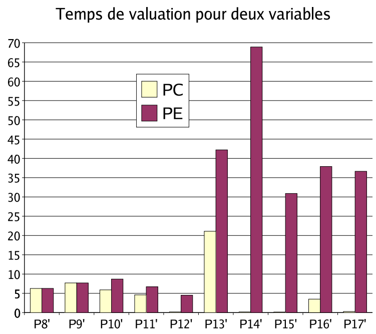
Tableau 3. *Expressions de chemins utilisées pour les mesures de la figure 7*

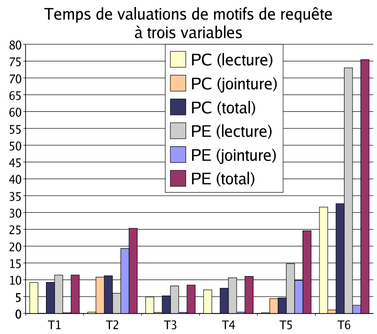
En général, le temps de lecture, de jointure et le temps total, sont encore bien plus petits pour PC que pour PE. Pour T_1, T_3, T_4 et T_6, le coût de la jointure est négligeable pour les deux, comparés au coût de lecture des données. Dans de tels cas, simplement utiliser un stockage PC suffit pour trouver des valuations de manière plus rapide. Pour T_2 et T_5, les jointures elles-mêmes prennent une part plus importante ; même avec le meilleur ordonnancement de jointures, la première jointure crée de nombreux résultats intermédiaires, sur lesquels la deuxième jointure doit passer plus de temps. Ce problème peut être réduit en appliquant des algorithmes de jointures multi-branches [BRU 02].

Effet des prédicats : dans l'absence des prédicats, la valuation d'un motif d'arbre peut lire moins de données que les plans de valuations considérés individuellement, puisqu'un accès séléctif à l'un d'entre eux peut être utilisé pour sauter des parties des entrées des autres (section 5.2).

Pour évaluer les bénéfices des prédicats, nous utilisons deux prédicats paramétriques :

– $f_{clust}(k)$ efface les premiers $(100 - k)$ % tuples de son entrée, et laisse passer les derniers k %.





Code de la requête	Requête	Document
T_1	for \$x in //person, \$y in \$x//name, \$z in \$x//watch . . .	XMark
T_2	for \$x in //asia, \$y in \$x//item, \$z in \$x//description	XMark
T_3	for \$x in //item, \$y in \$x//description, \$z in \$x//parlist	XMark
T_4	for \$x in //item, \$y in \$x//parlist, \$z in \$x//keyword	XMark
T_5	for \$x in //categories, \$y in \$x//listitem, \$z in \$x//text	XMark
T_6	for \$x in //article, \$y in \$x//title, \$z in \$x//year	DBLP

Figure 8. *Performance de la valuation sur des plans à 2 et 3 variables*

– $f_{unclust}(k)$ laisse passer $k\%$ de ses entrées, mais ces tuples sont répartis uniformément sur toute l'entrée. Si k est 10 par exemple, alors 1 sur 10 tuples successifs survit.

Nous prenons d'abord op_1 comme étant f_{clust}(IDScan(4), k) et op_2 comme étant IDScan(5) ; ceci équivaut à

for \$x in //person[@id$\geq\lambda$], \$y in \$x/name

où λ est (1-k/100)*N_4 (figure 9).

Ceci est le cas le meilleur pour STJ-FFD : la jointure arrive à éviter les premiers 100-$k\%$ tuples de op_2 dans une seule étape. De plus, dû au comprtement optimiste (section 5.2), STJ-FFD n'a jamais besoin de sauter sur les tuples qui restent dans op_2, puisqu'ils correspondent tous à op_1. Les gains de STJ-FFD sur STJ sont directement proportionnels à la partie de l'entrée qui a été court-circuitée, (100-k)*$|op_2|$.

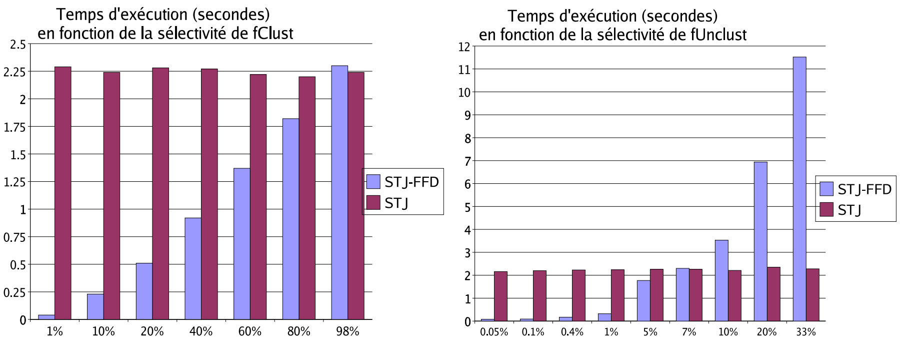


Figure 9. *Sauts sur le descendant : f_{clust} (à gauche) et $f_{unclust}$ (à droite)*

Nous considérons maintenant op_1 comme étant $f_{unclust}$(IDScan(4), k) et op_2 comme dans le cas précédent, équivalent à :

for \$x in //person[(@id mod 100)$\geq k$], \$y in \$x/name

Les résultats sont illustrés dans la figure 9. Ceci est le cas le pire pour STJ-FFD : après chaque paire qui correspond, il doit faire un saut en $O(log(|op_2|))$. Pour des séléctivités tres fortes, STJ-FFD garde ses avantages ; à partir de $k = 6$, le surcoût des sauts rend STJ-FFD pire que STJ. Nous avons enregistré des résultats très similaires sur STJ-FFA.

Nous dérivons l'heuristique suivante pour le cas général : utiliser des jointures qui sautent (STJ-FFA ou STJ-FFD) lorsque la taille estimée de l'une des entrées est très petite, p.ex. pour des requêtes de type point.

6.6. *Conclusions des expériences*

Le modèle PC est très compact, puisque son seul surcoût est dû à la présence d'identifiants structurels, qui sont assez petits. Un stockage organisé en PC peut être de l'ordre de 4 fois plus compact que l'approche PE basée sur des arbres B+.

Le modèle PC permet d'effectuer des valuations de variables qui peuvent être de plusieurs ordres de grandeurs plus rapides que PC, dû à ses connaissances structurelles précises. Ceci est la raison des bien meilleures performances de PC par rapport à son voisin proche PE utilisé dans [FIE 02, JAG 02, HAL 03].

Enfin, le modèle ordonné PC permet de sauter des parties inutiles des entrées pendant le traitement de jointures structurelles. Cet aspect est vrai aussi pour un modèle PC. Toutefois, le surcoût en termes de place de PC est plus petit, et l'usage d'un résumé de chemins permet de sauter des parties de l'entrée dans d'autres situations que celles précédemment identifiées.

7. Conclusion

Nous avons présenté le modèle partitionné par les chemins, un modèle compact et fragmenté pour le stockage des documents XML.

Les avantages fournis par ce modèle du point de vue du traitement de requêtes sont multiples. L'accès aux identifiants d'éléments se fait de manière très sélective, grâce à la fragmentation verticale du stockage. La valuation des variables multiples peut se faire de manière efficace en s'appuyant sur le résumé structurel du document. Enfin, les identifiants extraits à partir de ce modèle de stockage comportent des informations auxiliaires utiles pour le choix des opérateurs de jointure structurelle. Le modèle de stockage partitionné par les chemins a été mis en œuvre dans le système GeX ; ses performances ont été validées par une série d'expériences.

Les auteurs sont extrêmement reconnaissants à Alberto Lerner pour ses nombreux commentaires sur ce travail.

8. Bibliographie

[ABO 01] ABOULNAGA A., ALAMENDEEN A. R., NAUGHTON J., « Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Rome, Italy, 2001.

[ALK 02] AL-KHALIFA S., JAGADISH H., PATEL J., WU Y., KOUDAS N., SRIVASTAVA D., « Structural Joins : A Primitive for Efficient XML Query Pattern Matching », *Proceedings of International Conference on Data Engineering (ICDE)*, San Jose, CA, 2002.

[ARI 04] ARION A., BONIFATI A., COSTA G., D'AGUANNO S., MANOLESCU I., PUGLIESE A., « Efficient Query Evaluation over Compressed XML Data », *Proceedings of the Conf. on Extending Database Technology (EDBT)*, Heraklion, Crete, 2004.

[BER] « The BerkeleyDB database library », http ://www.sleepycat.com.

[BRU 02] BRUNO N., KOUDAS N., SRIVASTAVA D., « Holistic twig joins : Optimal XML pattern matching », *Proceedings of ACM SIGMOD Conference on Management of Data*, Madison, WI, USA, 2002.

[CHE 03] CHEN Z., JAGADISH H., LAKSHMANAN L., PAPARIZOS S., « From Tree Patterns to Generalized Tree Patterns : On Efficient Evaluation of XQuery », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, 2003.

[CHI 02] CHIEN S., VAGENA Z., ZHANG D., TSOTRAS V., « Efficient Structural Joins on Indexed XML Documents », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, 2002.

[FIE 02] FIEBIG T., HELMER S., KANNE C., MOERKOTTE G., NEUMANN J., SCHIELE R., WESTMANN T., « Anatomy of a native XML base management system », *VLDB Journal*, vol. 11, n° 4, 2002, p. 292-314.

[GAR 00] GARCIA-MOLINA H., ULLMAN J., WIDOM J., *Database System Implementation*, Prentice Hall, 2000.

[GOL 97] GOLDMAN R., WIDOM J., « DataGuides : Enabling Query Formulation and Optimization in Semistructured Databases », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[HAL 03] HALVERSON A., BURGER J., GALANIS L., KINI A., KRISHNAMURTHY R., RAO A., TIAN F., VIGLAS S., WANG Y., NAUGHTON J., DEWITT D., « Mixed Mode XML Query Processing », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, 2003.

[JAG 02] JAGADISH H. V., AL-KHALIFA S., CHAPMAN A., LAKSHMANAN L. V., NIERMAN A., PAPARIZOS S., PATEL J., SRIVASTAVA D., WIWATWATTANA N., WU Y., , YU. C., « Timber :a native XML database. », *VLDB Journal*, vol. 11, n° 4, 2002.

[LER 03] LERNER A., SHASHA D., « AQuery : Query Language for Ordered Data, Optimization Techniques, and Experiments », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, 2003.

[LIE 00] LIEFKE H., SUCIU D., « XMILL : An Efficient Compressor for XML Data », *Proceedings of ACM SIGMOD Conference on Management of Data*, 2000.

[MAR 03] MARIAN A., SIMEON J., « Projecting XML Documents », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, 2003.

[MCH 99] MCHUGH J., WIDOM J., « Query Optimization for XML », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, 1999.

[MIL 99] MILO T., SUCIU D., « Index Structures for Path Expressions », *Proceedings of the International Conference on Database Theory (ICDT)*, 1999, p. 277-295.

[NGO 03] NGOC T. T. D., GARDARIN G., « Evaluating XQuery in a full-XML mediation architecture », *Proceedings Journées Bases de Données Avancées*, Lyon, France, 2003.

[SAX] « The SAX project », www.saxproject.org.

[SHA 00] SHANMUGASUNDARAM J., SHEKITA E., BARR R., CAREY M., LINDSAY B., PIRAHESH H., REINWALD B., « Efficiently Publishing Relational Data as XML Documents », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000.

[TEU 03] TEUBNER J., GRUST T., VAN KEULEN M., « Bridging the GAP between Relational and Native XML Storage with Staircase Join », *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, 2003.

[WU 03] WU Y., PATEL J., JAGADISH H., « Structural join order selection for XML query optimization », *Proceedings of International Conference on Data Engineering (ICDE)*, Bangalore, India, 2003.

[XQU04] « The XQuery Language », www.w3.org/TR/xquery, 2004.

[ZHA 01] ZHANG C., NAUGHTON J., DEWITT D., LUO Q., LOHMAN G., « On Supporting Containment Queries in RDBMSs », *Proceedings of ACM SIGMOD Conference on Management of Data*, Santa Barbara, CA, USA, 2001.