Indexation de vues virtuelles dans un médiateur XML pour le traitement de XQuery Text

Clément Jamard*, Georges Gardarin*

Laboratoire PRiSM
Université de Versailles
78035, Versailles Cedex, France
prénom.nom@prism.uvsq.fr

Résumé: Intégrer le traitement de requêtes de recherche d'information dans un médiateur XML est un problème difficile. Ceci est notamment dû au fait que certaines sources de données ne permettent pas de recherche sur mot-clefs et distance ni de classer les résultats suivant leur pertinence. Dans cet article nous abordons l'intégration des fonctionnalités principales du standard XQuery Text dans XLive, un médiateur XML/XQuery. Pour cela nous avons choisi d'indexer des vues virtuelles de documents. Les documents virtuels sélectionnés sont transformés en objets des sources. L'opérateur de sélection du médiateur est étendu pour supporter des recherches d'information sur les documents de la vue. La recherche sur mots-clefs et le classement de résultat sont ainsi supportés. Notre formule de classement de résultats est adaptée au format de données semi-structurées, basé sur le nombre de mots-clefs dans les différents éléments et la distance entre les éléments d'un résultat.

1 Introduction

XQuery devenant le standard pour interroger XML, de nouveaux besoins apparaissent pour la recherche d'information. Buston et Rys (2003) spécifient des prédicats et fonctionnalités de recherche d'information à intégrer à XQuery, comme la recherche d'élément contenants des mots-clefs, le classement de résultats selon leur pertinence, la recherche basé sur des suffixes ou préfixes de mots. Un premier ensemble des fonctionnalités requises pour XQuery Text est défini par Buxton et Rys (2003). TexQuery, Amer-Yahia (2004), en est le langage précurseur.

Certaines des fonctionnalités citées précédemment, comme la simple recherche de mots-clefs, sont très communes et présentes dans la plupart des SGBD. Dans le cas de données distribuées, il faut d'abord recomposer les partitions avant de pouvoir effectuer une recherche sur le contenu ; d'importantes fonctionnalités souvent nécessaires aux applications ne sont pas faciles à implanter dans un système distribué. Le classement des résultats, les recherches conjonctives de mots-clefs, les recherches sur les racines de mots, leurs préfixes ou suffixes, sont difficilement réalisables car il faut auparavant recomposer les données dispersées.

De nombreux systèmes de médiation de données basés sur XQuery sont disponibles, comme BEA (2004), IBM DB2 (2004), Papakonstantinou (2003) ou XQuark (2004). Ils sont basés sur une architecture d'intégration au-travers de vues globales, et supportent un sous-ensemble du langage XQuery. A notre connaissance, aucun d'entre eux ne supporte le langage XQuery Text, alors que la plupart des applications d'intégration de données sont orientées vers la recherche d'information textuelle. Rappelons que l'objectif essentiel d'un médiateur est de fédérer les sources de données autour d'une architecture répondant au manque de capacité des sources. La plupart des SGBD (natifs XML ou relationnels) permettent la recherche sur un ou plusieurs mots-clefs dans des documents, mais la majorité de ces systèmes ont des capacités différentes. Google peut être interrogé en tant que collection virtuelle de documents XML. C'est un système très efficace pour effectuer une recherche par mots-clefs et classer les résultats suivant leur pertinence. Xyleme, Abiteboul (2002), un SGDB XML natif, permet lui aussi d'effectuer une recherche efficace sur des mots-clefs. Tous ces systèmes ont des capacités propres intéressantes, mais aucun ne permet de répondre à toutes les fonctionnalités de XQuery Text. Il existe donc un fort besoin d'intégration des fonctionnalités de ce langage dans les systèmes de médiation de sorte à pouvoir interroger d'une manière uniforme toutes les sources hétérogènes. Certaines capacités sont difficilement intégrables car elles sont spécifiques au système, comme classer les résultats suivant leur pertinence, où chaque système peur offrir ses propres fonctions.

Jusqu'à fin 2002, à l'Université de Versailles, nous avons été impliqué dans le développement de XQuark Fusion, un médiateur open source (XQuark). En 2003, nous avons extrait de ce projet un médiateur XML plus léger nommé XLive, qui est utilisé comme outil de recherche. XLive, Dang-Ngoc et Gardarin (2003), intègre et interroge des sources relationnelles ou XML en XQuery. Un large sous-ensemble de XQuery est reconnu, incluant les expressions FLWR et des requêtes imbriquées. Chaque source fédérée possède son propre adaptateur (wrapper) capable d'exécuter un sous-ensemble de XQuery. Le modèle d'exécution en flux est basé sur la XAlgébre, une algèbre dérivée de l'algèbre relationnelle, adaptée aux données semi-structurées. Cette algèbre représente les documents XML comme des tuples d'éléments désignés par des XPath référençant des arbres DOM codant du XML. Ces tuples sont appelés XTuple. Le médiateur évalue des plans composés d'opérateurs de la XAlgebre sur des ensembles de XTuples et recompose les résultats en XML.

Les données interrogées par XLive sont distribuées sur plusieurs sources. Un point important dans l'intégration des fonctionnalités de recherche d'information dans XLive est la gestion des capacités des sources. Des techniques d'indexations sont implémentées sur les sources non capables pour permettre des recherches sur mots-clefs dans des documents XML. De plus, comme les mots-clefs sont souvent recherchés dans des éléments précis, l'indexation de leur position est requise. Plusieurs techniques d'indexation ont été proposées dans le cas de systèmes centralisés pour la recherche d'éléments contenant des mots-clefs. Un aperçu de ces techniques est disponible dans Gardarin et Yeh (2004). Des techniques comme les T-index, Milo et Suciu (1999), APEX, Chin-Wan et al. (2002), A(k) indexes, Kaushik et al. (2002), D(k) indexes, Chen (2003) permettent de réduire la taille des index en indexant sur les chemins XPath utiles. De tels chemins peuvent être déterminés par des modèles spécifiques (T-index), ou en sélectionnant les chemins les plus fréquemment demandés (APEX), ou en réduisant la longueur des chemins à k (Kaushik et al.), ou encore en se basant sur la charge des requêtes (Chen). L'index Fabric, Cooper et al. (2001), évite de référencer tout les nœuds en codant les chemins des nœuds terminaux (feuilles) et en les

stockant dans un Patricia trie. Cette approche nécessite des extensions pour garder l'ordre des chemins et permettre la correspondance partielle des noms d'éléments.

Notre approche propose d'unifier les capacités des sources au travers de vues. Le médiateur définit une vue des données distribuées sur plusieurs sources, et permet son interrogation en XQuery Text. Le médiateur ne matérialise pas la vue pour éviter la réplication des données ; il indexe la position du texte dans les éléments structurant la vue. Nous proposons un système d'indexation efficace du contenu textuel de la vue qui repose sur un guide de la vue (applé ViewGuide), véritable résumé structurel de la vue, dérivé de la définition de la vue. Notre système est adapté pour la localisation de mots-clefs dans la vue, et la localisation des données sur les sources. XQuery/IR, Bremer et Gertz (2002), propose des techniques similaires de recherche d'information pour XQuery. Il est aussi basé sur une indexation adaptée à la structure arborescente de XML, et permet de résoudre des requêtes "tree pattern" (arbre de filtres applicables à des données XML) dans un système centralisé. L'originalité de notre approche est d'indexer des vues virtuelles et d'avoir une solution complète opérationnelle et efficace, comme le montre les premières mesures.

La suite de cet article est organisée comme suit. La section 2 présente le système d'indexation de vues proposé. La section 3 présente le traitement des requêtes sur les vues indexées et la méthode de classement des résultats suivant leur pertinence. Des résultats expérimentaux de notre système sont ensuite rapportés. La conclusion rappelle les contributions et introduit les travaux futurs.

2 Indexation textuelle de vues

La principale question est de savoir comment intégrer des méthodes de recherche de contenu sur des sources distribuées et hétérogènes. Les systèmes de médiation utilisent souvent les vues pour cibler la recherche sur les sources de données pertinentes. Pour combiner l'intérêt des vues avec la recherche d'information, nous avons décidé d'utiliser des vues semi-matérialisées : le contenu de la vue est indexé par le médiateur, mais n'est pas stocké.

2.1 Principes de base

Nous avons choisi d'indexer le contenu de la vue lors de sa création et de maintenir l'index lors des mises à jour ; la position des termes de la vue est mémorisée au niveau du médiateur, ce qui permet de répondre efficacement à des requêtes XQuery Text. L'index détermine indirectement l'adresse des éléments ; cela évite d'importants transferts de données entre les sources et le médiateur : seules les données pertinentes sont échangées. Cela évite également au médiateur de manipuler au travers d'opérations complexes de recherche d'information l'ensemble des données. Ainsi, gérer un index de vue compact et efficace est le but de notre approche pour éviter au médiateur ces opérations difficiles.

Les identifiants utilisés dans notre index référencent à l'aide de structures gérées par les adaptateurs des objets sur les sources. Ces structures permettent la localisation, l'extraction et la recomposition de données de la vue efficacement à partir des sources. Lorsqu'une source est mise à jour, celle-ci doit le reporter au médiateur afin de mettre à jour les identifiants dans l'index. Ceci est fait par un mécanisme de trigger ou bien par polling périodique de la source. Le mécanisme de report dépend de l'adaptateur de la source.

2.2 Position des termes dans la vue

Pour retrouver les éléments contenant un terme, le contenu textuel de la vue doit être indexé de façon précise. La position d'un terme dans la vue est identifiée par le document de la vue et le chemin (path) de l'élément le contenant. Nous proposons un système de d'identification d'élément pour coder cette position de façon compacte et unique.

2.2.1 Système de numérotation

Un élément de la vue est identifié par un identifiant de document global (IDG) et un identifiant d'élément (IDE) correspondant au chemin menant à cet élément.

Définition: Identifiant de document global (IDG). Identifiant numérique alloué par le médiateur identifiant un document de la vue.

Définition: Identifiant d'élément (IDE). Identifiant déterminant de manière unique un élément par son chemin.

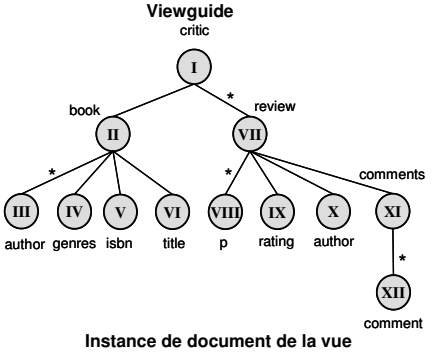
Pour coder un IDE, nous utilisons un guide résumant la structure de la vue.

Définition: Guide de vue (ViewGuide). Arbre représentant la structure commune de tous les documents de la vue. Les nœuds correspondent aux noms d'éléments ou d'attributs. Les liens entre les éléments sont marqués avec leur cardinalité simple ou multiple. Chaque chemin distinct des documents de la vue est représenté une et une seule fois dans le guide.

Définition de la vue en XQuery

```
for $b in collection("catalog")/book
return
<critic>
  <book>
      {   for $a in $b/author
          return <author> { $a/text () } </author>
      }
      { $b/genres }
      <isbn> { $b/@isbn/text () } </isbn>
      { $b/title }
  </book>
  { for $rev in collection("review")/review
    where $b/@isbn = $rev/book/@isbn
    return
      <review>
        {   for $p in $rev/book/p
            return <p> { $p/text () } </p>
        }
        { $rev/book/rating }
        { $rev/book/author }
        <comments>
        {   for $c in collection("comments")/comment
            where $c/@id = $rev/@id
            return <comment> { $c/text () } <comment>
        }
        </comments>
      </review>
  }
</critic>
```

Viewguide



Instance de document de la vue

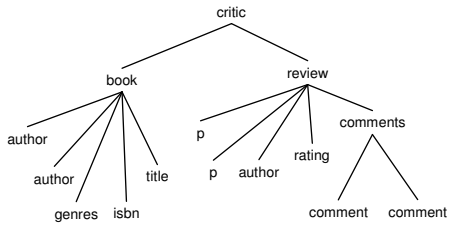


FIG. 1 – *Requête XQuery de définition de la vue avec son ViewGuide.*

Le ViewGuide est quelque peu similaire au DataGuide (Widom et al.), mais il diffère par les points suivants : (i) c'est uniquement un résumé structurel des documents de la vue ; (ii) il

est dérivé de la définition de la vue (c'est-à-dire de la requête définissant la vue) ; (iii) ses liens sont annotés avec la cardinalité (multiple ou non) des éléments. Le ViewGuide permet d'attribuer un identifiant d'élément (IDE) à chaque élément d'un document de la vue. Chaque élément correspond à un identifiant numérique unique, déterminé par un parcours préfixe de l'arbre. La figure 1 présente un exemple de ViewGuide. Un élément critic d'un document de cette vue ne contient qu'un élément book (monovalué) mais peut englober un ou plusieurs éléments review (multivalué). La création du guide impose que la requête définissant la vue (figure 1) décrive complètement la structure des documents dans sa clause Return.

Un IDE est composé de la façon suivante :

- Un préfix, l'identifiant correspondant au chemin dans le guide de vue.
- Un suffixe, regroupant les cardinalités de tous les éléments multivalués traversés de la racine à l'élément concerné.

Pour identifier le chemin correspondant à *critic/review/p*, le ViewGuide traverse les éléments I, VII puis VIII. Seul le dernier identifiant (VIII) est utile car le chemin est unique dans le ViewGuide. Le document exemple de la figure 1 possède un élément *review* contenant deux éléments *p*, ce qui correspond au chemin *critic/review[1]/p[2]*. L'identifiant de cet élément sera donc codé par VIII(1,2). Le suffixe (1,2) associé à l'identifiant de chemin VIII permet d'identifier de manière unique le deuxième élément *p* du premier élément *review*. Le suffixe n'est utilisé que dans le cas d'éléments multivalués ; un élément monovalué n'aura pas de suffixe, par exemple VI code le chemin *critic/book/title*.

Pour indiquer la position d'un élément dans la vue, l'identifiant de document (IDG) est associé à l'identifiant de chemin (IDE). Dans la vue de la figure 1, chaque élément *critic* est un document. L'élément *author* de la seconde *review* du quatrième document de la vue est identifié par <4-X(2)>. Un tel couple identifie de manière unique un élément dans la vue.

Pour coder des chemins multiples, nous utilisons des patterns d'identifiant d'éléments permettant de spécifier un ou plusieurs éléments multivalués traversés. Le chemin *critic/review/p[1]* ne spécifie pas quel élément *review* choisir. Il se code avec le pattern VIII(*,1), correspondant au premier *p* de n'importe quel *review*. Les patterns d'identifiants sont utilisés pour le traitement de requête.

Définition: Pattern d'identifiants (IDEP). IDE ayant pour suffixe des *, signifiant que tout élément est valide.

2.2.2 Index des mots

Le médiateur gère une liste inversée des termes importants, donnant pour chacun sa position dans la vue virtuelle.

Définition: Index des mots (Word index). B-Tree de termes donnant pour chaque terme la liste des identifiants où apparaît ce terme dans la vue.

IDG	Préfixe	Suffixes
120	IV	-
120	VI	-
120	VIII	(1,2) (2,3) (2,5)
120	XII	(1,1)

IDG	Préfixe	Suffixes
120	VIII	(1,4) (3,5)
120	XII	(2,4)
121	VI	-
121	VIII	(3,4) (4,1)

TAB. 1 – *Tables de position, record1 et record2.*

La position d'un terme est repérée par un couple <IDG-IDE>. L'index des mots permet de retrouver toutes les positions d'un mot dans la vue. Lors de la création de la vue, le médiateur crée et insère les entrées dans cet index. Les mises à jour des sources sont répercutées.

Les entrées de l'index des mots sont des paires (terme, table de position). Une table de position est une table référençant les IDG, IDE préfixe et IDE suffixes. Pour chaque IDG on retrouve une ligne par préfix IDE et l'ensemble des suffixes ordonnés. Cette organisation des entrées illustrée dans le tableau 1 est utilisée par l'algorithme de traitement des requêtes.

2.3 Position dans les sources de données

Une structure appelée "Source Map" maintient un mapping entre un document (IDG) et les sources contenant les données composant ce document. Ces données sont considérées comme des documents locaux référencés par un identifiant. Cet identifiant est associé à une opération d'extraction.

Définition : Identifiant de document local (IDL). Identifiant numérique alloué par un wrapper permettant de récupérer les données correspondantes sur une source.

Lors de la création de la vue, chaque wrapper de source contenant des données pour la vue reçoit une requête pour extraire des données. Les wrappers fournissent ces données au médiateur qui construit les résultats suivant la définition de la vue. Pour chacune de ces données renvoyées, un IDL est créé.

La correspondance IDL vers objet local dépend du wrapper. Pour un wrapper fichier, l'IDL peut être l'URI du fichier. Pour des bases XML, ce peut être un identifiant de document, pour des bases relationnelles, une référence à une requête SQL/XML ou XQuery permettant la transformation de tables relationnelles en XML. A partir d'un IDL, le wrapper est capable d'interroger la source pour renvoyer les données correspondantes. Le médiateur recompose alors le document suivant la définition de la vue en récupérant les données.

Definition: Source Map. Structure de mapping entre un IDG et un ou plusieurs IDL.

Par exemple, si les données *book*, *review* et *comment* de la vue sont réparties sur trois sources, l'IDG du document exemple aura alors quatre IDL (correspondant successivement à un *book*, une *review* et deux *comment*).

3 Traitement de requête

L'algorithme de traitement de requêtes retrouve les entrées de l'index correspondant à une recherche textuelle (liste de mots-clef). Les parties correspondant aux documents sélectionnés sont alors extraites pour recomposer les documents. Nous détaillons dans cette partie comment s'effectue une recherche d'éléments contenant des mots-clefs et comment étendre cette recherche aux fonctionnalités XQuery Text. Nous introduisons aussi notre système de classement de résultats suivant leur pertinence.

3.1 Trouver les éléments pertinents

Une recherche sur plusieurs mots-clefs en XQuery Text repose sur deux paramètres : l'espace de recherche et la liste des mots-clefs. L'espace de recherche définit l'ensemble d'éléments dans lesquels effectuer la recherche, typiquement un XPath. Nous nous

concentrons ici sur des recherches conjonctives de mots-clefs, les éléments devant donc contenir tous les mots-clefs. Les prédicats dans les requêtes sont de la forme A1/A2…/Am [. ftcontains k1 && k2 … kn], ou les Ai sont des noms d'éléments et ki les mots-clefs. Les expressions régulières avec joker * ou wildcard // sont autorisées pour les Ai. La requête « retourner les éléments book de la vue critic ayant un élément review contenant les termes 'XML' et 'databases' » s'exprime en XQuery Text comme suit :

```
for $b in collection("critic_view")/critic
where $b/review[. ftcontains "XML" && "databases"]
return { $b/book }
```

```
    ALGORITHM INTERSECT
    INPUT       : List(n)
                      n list of keyword position
    VARIABLES : intersect, current_id, test_id, tmp_res
                      type:identifier<IDG-IDE>
                search_space
                      type:IDEG filter
1.              result
2.                    type:list of result
3.  current_id = List(0).nextElement(search_space)
4.  WHILE( List(0).hasNextElement(search_space) ) DO
5.  {
6.      intersect = NULL;
7.      FOR EACH LIST DO
8.      {
9.          test_id = list().nextElement(search_space)
10         /* Aucun élément dans le même document
11         Itération sur l'élément suivant */
12         WHILE (  test_id.IDG > current_id.IDG )
13         {
14             current_id = test_id;
15         }
16.
17. ELSE
18.         {
19.             /* Trouve le prochain élément avec le même IDG */
20.             WHILE( test_id.IDG < current_id.IDG ) DO
21.                 test_id = list().nextElement(search_space);
22.             /* Intersecte deux IDE dans l'espace de recherche */
23.             tmp_res = intersection( test_id, current_id, search_space );
24.             /* intersection valide -> passe à la liste suivante */
25.             /* intersection invalide -> passe à l'élément suivant */
26.             IF( tmp_res != NULL && !tmp_res.descendantOf(result.last))
27.                 intersect = tmp_res;
28.             ELSE
29.                 intersect = NULL;
30.                 IF( current_id.IDE.suffix > test_id.IDE.suffix )
31.                     current_id = list().nextElement(search_space);
32.                 break;
33.         }
34.     }
35.     /* Intersection valide entre toutes les listes, résultat validé */
36.     if(intersect != NULL)
37.         result.add(intersect);
38.         last_res = intersect;
39.         current_id = list().nextElement(search_space);
40. }
```

FIG. 2 – *Algorithme d'intersection*

La recherche des éléments résultat s'effectue en 3 étapes :

1. Déterminer l'espace de recherche ;
2. Calculer l'ensemble des éléments contenant chacun des mots-clefs ;
3. Extraire les données des sources et recomposer les résultats.

L'espace de recherche se définit en utilisant les patterns d'identifiants d'élément. Le NIP est dérivé de l'expression XPath correspondant au prédicat de la requête.

Le XPath définissant l'espace de recherche dans la requête exemple est *critic/review*. Il correspond au pattern VII(*). La recherche des mots-clefs s'effectue sur l'élément *review* et tous ses descendants ; les mots-clefs peuvent se trouver parmi les éléments *comments*, *comment*, *p*, *author* ou *rating*. Pour retrouver les identifiants correspondant au descendant d'un élément, une matrice booléenne indique la relation ancêtre/descendant entre deux identifiants. Un élément $C_{ij} = 1$ si *i* est un ancêtre de *j* dans le guide de la vue. Cette matrice fournit une méthode rapide pour retrouver la relation entre deux éléments. L'espace de recherche *critic/review* est ainsi déterminé par les patterns allant de VII(*) à XII(*).

Le médiateur interroge l'index des mots pour calculer l'ensemble des entrées contenu dans l'espace de recherche pour chaque mot-clef. L'ensemble des éléments résultat est

l'intersection de ces entrées. L'opération d'intersection de deux entrées détermine si l'ancêtre commun des deux éléments appartient à l'espace de recherche.

Pour réduire le nombre d'opérations d'intersections, les intersections ne sont calculées que pour des éléments respectant les conditions suivantes (dans l'ordre) :

1. égalité des identifiants de documents. Une intersection d'éléments de deux documents différents est nulle.
2. le préfixe des IDE correspond à un descendant du NIP espace de recherche. Les entrées hors de l'espace de recherche ne sont pas considérées.
3. le suffixe de l'IDE intersecté correspond à une intersection déjà calculée.

La figure 2 décrit l'algorithme d'intersection. Il calcule chaque intersection valide entre deux listes d'identifiants dans un espace de recherche. La première condition est vérifiée aux lignes 12 et 20-21 en sélectionnant des entrées de même document. La seconde est vérifiée à chaque accès aux identifiants aux lignes 3, 9,21, 31. L'élément est choisi parmi les lignes des tables de position correspondant à l'espace de recherche. La troisième est appliquée à la ligne 26, en vérifiant si l'intersection n'a pas déjà été calculée.

Comme les entrées sont ordonnées par suffixe de IDE dans les tables de position, aucune intersection ne peut être oubliée. La condition de la ligne 30 sélectionne donc toujours le plus petit IDE pour l'intersection.

Cet algorithme peut être appliqué aux deux tables de position du tableau 1 avec l'espace de recherche VII(*). Pour l'identifiant 120, seules les entrées des lignes VIII et XII sont sélectionnées. Les intersections valides entre $XII(1,1)_{record1}$ et $VIII(1,4)_{record2}$, $VIII(2,3)_{record1}$ et $XII(2,4)_{record2}$ sont VII(1) et VII(2). Toutes les autres intersections ne sont soit pas dans l'espace de recherche, soit déjà calculées. Le même traitement est fait sur les autres patterns.

3.2 Supporter les fonctionnalités XQuery Text

3.2.1 Recherche plein texte

XQuery Text permet des recherches au niveau d'un élément (seul le chemin depuis la racine du document identifie cette position) mais aussi relativement aux autres mots dans un élément (le chemin et la position du mot dans le texte identifient cette position). L'index présenté précédemment permet de répondre à des requêtes sur la position des mots au niveau élément. Les autres fonctionnalités, comme la distance entre les mots, l'ordre ou le classement des résultats, nécessitent de connaître la position des termes plus précisément. Les entrées de l'index peuvent être complétées pour ajouter la position du mot par rapport au début du texte de l'élément. Les fonctionnalités pour les index d'élément et d'élément/position sont données dans le tableau 2.

Index d'élément	Index d'élément/position
[. ftcontains "information" && "data"]	[. ftcontains "xml" && "data" with distance *n*]
[. ftcontains "data" with stemming]	[. ftcontains "xml" && "data" ordered]
[. ftcontains "data" with thesaurus "th"]	[. ftcontains "data information" starts with]
[. ftcontains "inform*" with regex]	[.ftcontains "xml data" exact content]
[. ftcontains "data" without content .//p]	[.ftcontains "xml" && "data" mild not "xml data"]

TAB. 2 – *Prédicats XQuery Text calculable avec les index.*

3.2.2 Classement des résultats

Une méthode de classement associe un poids, basé sur la pertinence, à chaque résultat. Le médiateur doit classer les résultats provenant de plusieurs sources et les regrouper pour les renvoyer dans l'ordre du classement. L'architecture proposée permet de pré-calculer le poids de chaque résultat avant de recomposer les données ; le calcul est réalisé lors de l'interrogation de l'index des mots. La formule de classement doit être précise mais aussi calculable avec les informations contenues dans l'index.

Le poids d'un résultat est la somme des poids de chaque élément du résultat contenant un ou plusieurs mots-clefs. Notre approche est basée sur la spécificité de chaque résultat. Cette méthode donne plus d'influence aux éléments proche de la racine du résultat. En effet, les mots proches de la racine sont plus importants que ceux dans des éléments plus profonds de l'arbre résultat. Cette méthode reste encore assez simple puisqu'elle ne prend pas en compte la position des mots relativement entre eux.

Les éléments contenant plusieurs mots recherchés voient leur influence également augmenter. Le pourcentage de mots-clefs présent dans l'élément par rapport à l'ensemble des mots-clefs recherchés est un facteur polynomial ajustant le poids d'un élément.

Finalement, la formule suivante calcule le poids d'un élément:

$$We = \frac{Ni^{\beta}}{N} \sum_{i=0}^{n} \left(\frac{Wi}{(1+\alpha)^{d}} \right)$$

W_i est le poids du mot recherché k_i, basé sur le tf.idf du mot, N est le nombre total de mots recherchés dans le prédicat. Ni est le nombre de mots recherchés présents dans l'élément et d est la distance entre l'élément et la racine du résultat (nombre de liens). La constante α permet de faire varier l'influence de la distance à la racine. β est un facteur polynomial qui permet d'augmenter l'influence des éléments contenant plus de mots recherchés. Le poids total d'un résultat est la somme des poids de chaque élément contenant des mots-clefs.

Cette formule modulable peut s'adapter au besoin de l'utilisateur. Elle peut être étendue ou remplacée. Ainsi le médiateur peut intégrer d'autres formules reposant sur les informations de l'index des mots. La formule est ajustable suivant l'application.

D'autres systèmes proposent des solutions concrètes pour classer les résultats d'une requête de recherche de mots-clefs. XRANK, Lin et al. (2003), propose une formule calculée suivant le nombre d'arc entrant et sortant (inter et intra document) d'un élément. Ce système utilise comme métrique de proximité des termes la fenêtre minimum contenant les termes, facteur qui reste trop global ne tenant pas compte de la structure du résultat. La distance est aussi prise en compte comme notre approche, ou les éléments les plus éloignés ont moins d'importance. D'autres systèmes comme XXL, Anja et Gerhard (2002), se base sur un opérateur d'imprécision donnant un degré de similarité entre la structure d'un résultat et la demande de la requête. XIRQL, Norbert et Kai (2001), découpe les documents en objets et recherche ces objets suivant leur pertinence de contenu.

4 Retour d'expériences

Nous avons testé les performances du système sur trois jeux de données. Les données présentées dans le tableau 3 sont stockées dans des SGBD XML. La taille des données est la

taille de la vue composée suivant la structure de la vue *critic*. Nous avons mesuré le temps de recherche pour trois prédicats :

(q01) critic/review [. ftcontains "k_1"&& … "k_n"]
(q02) critic [. ftcontains "k_1" && … "k_n" without content .//title]
(q03) critic [. ftcontains "k_1" && … "k_n"]

Jeux de données			*Exécution (ms)*		*Intersection (ms)*		
	Docs	Byte Size	View	Mediator	i=5	i=15	i=25
DS1	100	607 855	1042.1	2917.6	1.6	3.1	4.6
DS2	250	1 556 052	1976.5	5969	6.3	15.6	21.9
DS3	500	3 029 990	5949	18501	17.2	65.6	86

TAB. 3 – *Jeux de données. Evaluation puis intersection de q01.*

Les requêtes renvoient les documents contenant les mots-clefs ki. La recherche est définie sur plusieurs parties de la vue ; q01 sur les éléments *review*, q02 sur *review* sans *title*, et q03 sur l'ensemble. Le nombre de mots-clefs varient entre 2 et 26. Les temps d'exécution des requêtes sont une moyenne de dix exécutions sur un Pentium 4 avec 512Mo de mémoire.

Le second sous-tableau du tableau 3 donne les temps d'exécution de q01 pour 5 mots-clefs. Pour chaque jeu de données, l'exécution est réalisée avec une vue indexée, et avec un opérateur de recherche (le médiateur s'occupe de l'opération de recherche dans chaque document). Le temps présenté pour la recherche avec l'index inclut le temps de recherche dans l'index (Word Index et Source Map), l'extraction et la recomposition des résultats par le médiateur. Pour l'opérateur de recherche, le temps inclut la recomposition des résultats suivant la définition de la vue, puis l'application du prédicat de recherche de mots-clefs par un opérateur de sélection. Comme prévu, l'exécution en utilisant l'index est plus efficace car seulement les résultats pertinents sont extraits des sources. Pour chaque jeu de donnée, un ratio de 3 est obtenu pour des requêtes sélectionnant 66% de l'ensemble des documents.

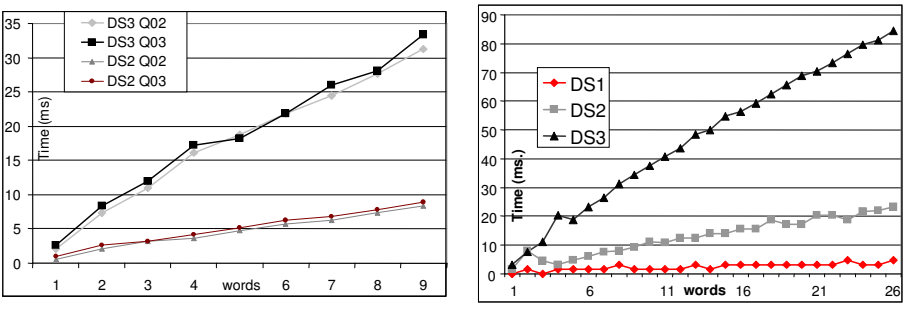


FIG. 3 – *Temps d'évaluation de la recherche dans l'index.*

Le temps de recherche dans l'index (algorithme d'intersection de listes) est présenté dans le dernier tableau du tableau 3, pour la requête q01. Comparé au temps d'exécution total de la requête (tableau précédent), la recherche dans l'index représente moins de 1%. Ces tests

préliminaires montrent la validité de l'approche : moins de données sont transférées, et l'opération de recherche au niveau du médiateur est plus rapide avec un index.

Le premier graphe de la figure 3 illustre les temps mis par l'algorithme d'intersection pour retrouver les entrées pertinentes pour les requêtes q02 et q03. On remarque que q02 s'exécute plus rapidement que q03 ; ceci est dû aux sélections faites dans l'algorithme d'intersection, l'espace de recherche de q02 étant plus restreint que celui de q03.

Le second graphe de la figure 3 présente les temps de recherche dans l'index pour q01. Pour chaque jeu de donnée, le temps de recherche augmente linéairement lorsqu'un nouveau mot-clef est ajouté à la recherche. Finalement, pour des requêtes basiques (moins de 10 mots-clefs), le système est efficace et le temps de recherche dans l'index est négligeable.

5 Conclusion

Dans cet article, nous avons présenté l'intégration de XQuery Text dans un médiateur XML. La principale difficulté est d'intégrer des sources ne répondant pas à ces capacités. Pour cela nous proposons d'utiliser des vues indexées pour permettre d'intégrer ces fonctionnalités à ces sources. Le médiateur indexe les vues en utilisant un résumé structurel de la vue. Ce guide permet de coder la position des éléments (XPath) de la vue pour en indexer le contenu. L'opérateur de recherche de mots-clefs utilise des algorithmes basés sur ce système d'identifiant pour l'exécution de requête XQuery Text. Le système intègre une formule de classement adaptée à la structure arborescente des résultats XML.

Il reste d'autres aspects important à aborder dans la gestion des capacités des sources. Lorsqu'une source reconnaît une partie de XQuery Text, les vues construites devraient prendre en compte cette capacité et limiter l'indexation aux sources non capables en distribuant le traitement de la requête aux sources capables. Le classement de résultats semble simple pour une vue ou une source, mais le classement global doit être testé plus en détails, notamment en comparaison avec d'autres formules dans de vraies applications. Le dernier aspect à préciser est la gestion des mises à jours sur les sources et dans l'index, qui doit être mis à jour lors de l'insertion ou de la suppression d'objets dans les sources.

Références

Abiteboul S., S. Cluet, G. Ferran et M.C. Rousset: "The Xyleme project", Computer Networks 39(3): 225-238 (2002)

Amer-Yahia S., C. Botev, J. Shanmugasundaram : "TeXQuery: A Full-Text Search Extension to XQuery", WWW'04

BEA: "Liquid data for WebLogic 1.1, 2004, http://e-docs.bea.com/liquiddata/docs11/

Bremer J.M., M. Gertz : "XQuery/IR: Integrating XML Document and Data Retrieval", WebDB 2002.

Buxton S., Rys M. Editors, "XQuery and XPath Full-Text Requirements", W3C Working Draft 02 May 2003, http://www.w3.org/TR/xquery-full-text-requirements/

Chen Q., A. Lim and K.W. Ong : D(k)-index: An adaptive structural summary for graph-structured data. In Proc. of SIGMOD, 2003.

Chung Chin-Wan, J. Min and K. Shim: "APEX: an adaptive path index for XML data", SIGMOD Conference 2002: 121-132

Cooper B., N. Sample, M.J. Franklin, G.R. Hjaltason and M. Shadmon :" A Fast Index for Semistructured Data.", VLDB 2001: 341-350

Dang-Ngoc T.-T. et G. Gardarin : "Federating heterogeneous data sources with XML", In Proc. of IASTED IKS Conference, pages 193-198, Scottsdale, USA, Nov. 2003.

Fuhr Norbert and K. Großjohann: "XIRQL: A Query Language for Information Retrieval in XML Documents". SIGIR 2001: 172-180

Gardarin Georges et L. Yeh: "Treeguide Index: Enabling Efficient XML Query Processing", Bases de Données Avancées, Montpellier, Octobre 2005

Guo Lin, F. Shao, C. Botev, J. Shanmugasundaram : XRANK: Ranked Keyword Search over XML Documents. SIGMOD Conference 2003: 16-27

IBM: "DB2 Information Integrator for Content", 2004, http://www-306.ibm.com/software/data/eip/

Kaushik R., P. Shenoy, P. Bohannon and E. Gudes : Exploiting local similarity for indexing paths in graph-structured data. In Proc. of ICDE, 2002.

Tova Milo and D. Suciu: "Index Structures for Path Expressions", ICDT 1999: 277-295

Papakonstantinou Y. et al.: "XML queries and algebra in the Enosys integration platform", Data Knowl. Eng. 44(3): 299-322 (2003)

Theobald A and G. Weikum : "The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking". EDBT 2002: 477-495

Widom J. et. al.: "Lore, a DBMS for XML", http://www-db.stanford.edu/lore/

XQuark: "The XQuark project: open source information integration components based on XML and XQuery", 2004, www.xquark.org

Summary

Supporting full-text query in an XML mediator is a difficult problem. This is because most data-sources do not provide keyword search and ranking. In this paper, we report on the integration of the main functionalities of the emerging XQuery Text standard in XLive, a full XML/XQuery mediator. Our approach is to index virtual documents in views. Selected virtual documents are on demand mapped to data source objects. Thus, the mediator selection operator is efficiently extended to support full-text search on views. Keyword search and result ranking are integrated. We rank results using a relevance formula adapted to tree results, based on number of keywords in elements and distance from searched nodes.