Efficient XML Query Rewriting over the Multiple XML Views

Jun Gao, Dongqing Yang, Tengjiao Wang

The School of Electronic Engineering and Computer Science,Peking University ,100871 Beijing, China

{gaojun, dqyang, tjwang }@ pku.edu.cn

Abstract XML query rewriting in the distributed computing environment receives high attention recently. Different from existing work, we focus on the query rewriting in the cases that there are multiple views at the client side. We design two data structures to manage the multiple XML views. MPTree is constructed from the main path of the XML views to generate the candidate query rewriting plan. PPLattice is constructed from the predicate sub trees of the XPath views to validate the candidate query rewriting plan. Based on MPtree and PPLattice, the query rewriting plans search space over multiple views is pruned significantly and the high cost of the query containment can be reduced.

Keyword XML, Rewrite, View, Query

1. Introduction

With XML becoming the information representation and exchange standard in the Internet, XML is used in the distributed computing environment increasingly. Due to the high cost of the network transfer and the complexity of the XML query evaluation, the cached XML views at the client side play important roles in the query optimization. When a user submits a query to the remote server, system attempts to answer the query using the views at the client side directly[2]. In this way, the response time in the distributed computing environment can be reduced significantly.

The query rewriting problem has been studied extensively in the relational database context. Due to the complex features supported, the XML query rewriting faces more challenge than the relational query. The current study [9,10] shows that the XML query containment – one of the important steps in the XML query rewriting - is Co-NP problem even when XPath supports the limited features including {//,/,*,[]}.

Many methods are proposed to handle the XML query rewriting over the XML view. The query rewriting generation at the server side are studied [1] to generate the sound but incomplete plan over XPath {//,/,|,*,[]} in a polynomial time if the index can be regarded as the special case of the XML materialized view. The XPath query rewriting problem at the client side [2] is studied to show the special cases of XPath with restrictive features when the query rewriting plan is sound and correct.

Due to the different frequent query patterns, there may be multiple XML views cached in the client side. The XPath views selection problem has been discussed [3] over hundreds of views. Although the method proposed in [2] can be extended to support multiple views via combining the query rewriting results over each view with UNION operation, the whole process surfers the performance problem. Each view has to be handled to generate the candidate plan separately. In addition, the time consuming rewriting plan validation process has to be implemented for each view. We use the following example to demonstrate the challenges in the query rewriting over the multiple views.

Example 1: Given a NASA XML database [12] and a query that looks for the *partNumber*(*part*) of a *dataset*(*data*) in which the *abstract(abs)* is available in at least one *descriptions(dess)* that has nested *description(des)* is writing in XPath:

q_1:/*data*[*dess* [./*abs*][./*des*]/*title*/*part*;

Assuming the following three materialized views are defined over the database:

v_1: /*data* [*dess*[./*abs*/*para*][./*des*][*detail*]/*title* [*foot*];

v_2: /*[*dess* [./*abs*/*para*][.//*des*]]/*title*[*foot*];

v_3: /*data* [//*dess*[./*abs*/*para*][./*des*][*detail*]/*title* [*foot*];

Notice that the XML views may share the same sub paths. For example, v_2 and v_3 share the same sub part /*data*/*title*/*foot*. In the case that the same sub path appears in the main path, there are may be redundant cost in the query rewriting plan generation. In the case that the same sub path appears in the predicate sub tree, there are may be redundant cost in the query rewriting plans validation.

In order to prune the query rewriting plans space over the multiple XML views and improve the efficiency of the query rewriting plan generation, we propose a framework to organize the multiple XML views in a more manageable way. More specifically, our contributions can be summarized as follows:

- We propose MPTree to manage the main paths of the different XML views. Based on MPTree, we generate the query rewriting candidate plan efficiently and reduce the redundant computation among them.

- We propose PPLattice to organize the predicate sub trees from each XML views. Each node in PPLattice represents single path from the root node to the leaf node in the predicate sub tree. The edge in the PPLattice represents the containment relationship between the single paths which is pre-computed. Based on the PPLattice, we can locate more restrictive single XPaths for the XPath query in the XML views.
- After the pruning strategies over the MPTree and PPLattice, we propose a sound method to validate the query rewriting plans as a whole. Compared with the existing methods, only the views passing the requirement of the MPTree and PPLattice will be considered in the whole tree validation.

The rest of the paper is organized as follows: we introduces the background knowledge and define the problem in section 2; we propose MPTree and PPLattice in section 3; we discuss the whole process to generate the rewriting plan over the multiple views based on MPTree and PPLattice in section 4; related work is discussed in section 5; the paper is summarized and the future work is discussed in section 6.

2. Background Knowledge and Problem Definition

2.1. XPath

XPath is a basic mechanism to select nodes in the XML tree. It can be represented by the XPath tree pattern [9], which can be defined as:

Definition 1: (XPath tree pattern). the XPath tree pattern can be represented by a *query pattern* $Q=(N,E,r)$, where N represents the set of element nodes, E represents the relationship between the elements in N, for each $e \in E$, $type(e)=AD$ (short for ancestor/descendant) or PC (short for parent/child); r denotes the root node of the tree; Suppose L denote to the leaf nodes of the tree, given an element $n \in L$, n is called *return node* if n is the root element of the query result; the path from the root node to the return node is called the *main path*; The path from the root node to any leaf node is called the *single XPath query*.

The following are the tree patterns of the query q_1 and view v_1, v_2, v_3 in Example 1.

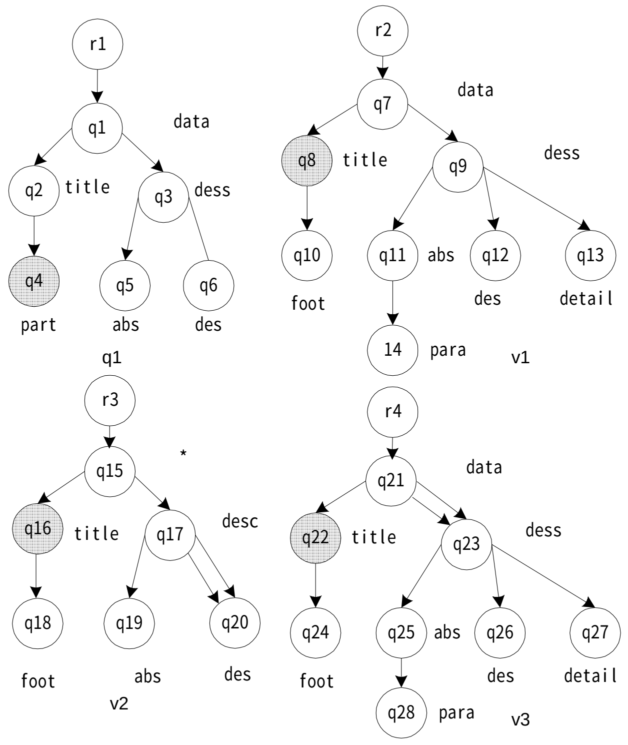

Fig. 1: XPath Tree Pattern for q_1, v_1, v_2, v_3

Definition 2: (The restrictiveness of XPath) Given two XPath p_1 and p_2, if we build the nodes mapping M from the XPath tree p_1 to the XPath tree p_2, for each sub path p_{11} from n_{11} to n_{12} in p_1, the mapped sub path $M(p_{11})$ from $M(n_{11})$ to $M(n_{12})$ in p_2, $M(p_{11})$ is contained in p_{11}, we call p_2 is more restrictive than p_1.

Given XPath p_1 and p_1, the restrictiveness comes from two aspects. In one case, the length p_2 is much longer than that of p_1. In another case, the path set P for p_2 is a sub set of those of the p_1. In both cases, we can establish the containment mapping from p_1 to p_2. We also notice that the containment is transitive. That is, if p_1 is more restrictive than p_2, and p_2 is more restrictive than p_3, p_1 is more restrictive than p_3.

2.2. View-based Query Answering

The problem of answering queries using the data in the existing materialized views has been extensively studied for the past decade. The related problems can be formally defined as follows [8]:

Definition 3: (Query rewriting plan). Given a query q over a database D, and a set of view $V=\{v_1,...v_n\}$ over the same database D, find a query q_1 which can be evaluated on the view set V, where $q_1(V) \subseteq q(D)$, $q_1(V)$ denotes that the results of query q_1 evaluated on the view set V; $q(D)$ denotes that the results of q evaluated on the data D;

Two notations, query containment and query equivalent, are used to validate the query rewriting plan.

Definition 4: (Query Containment and Equivalence). A query q_1 is said to be *contained* in a query q_2, denoted by $q_1 \subseteq q_2$, if for any database D, the results computed from q_1 are a subset of those computed for q_2, i.e., $q_1(D) \subseteq q_2(D)$. A query q_1 is said to be equivalence of a query q_2, if for any database D, $q_1(D) \subseteq q_2(D)$ and $q_1(D) \supseteq q_2(D)$.

Definition 5: (XML containment mapping). Given a XPath query tree pattern q, a XPath view tree pattern v, view is more restrictive than the query if we can establish the mapping from the query to the view, where the sub path p_1 in the view is more restrictive than $M(p_1)$ in query.

From the above definition, we notice that for each single XPath in the query, we can find at least one more restrictive single XPath in the view. Such requirement can be used as one of the heuristic information in the query rewriting. That is, if we can not find a more restrictive single XPath in the view v for the XPath query q, the view v can not be used to answer the query q without further validation.

2.3. The Problem Definition

With the above definitions, the problem in this paper can be defined as follow:

Given an XML Database T, XML structure schema D, a set of pre-defined materialized views V over the schema D, and an XPath{[],//,,/} query Q, , how to generate valid query rewriting plan answering plan for views set V efficiently?*

3. Query Rewriting over the Multiple Views

3.1. System Infrastructure

Recall the basic process to generate the query rewriting plan over the single XML view first. Given one XPath view and one XPath query, we try to match the main path of the query against the main path of the view. If matched, we generate the candidate query rewriting plan, which can be further validated with the consideration of the predicate sub-tree.

The following figure illustrates the mapping of the landmark nodes from the XPath query tree pattern to the XPath view tree pattern. One node n in the main path of the XPath called *the separate node* divides the main path into a *prefix path* and a *suffix path*. The prefix path is less restrictive than the main path of the XML view. Accordingly, the branching predicate associated with the original XPath expression is divided into two sets: *the outer predicates* – the predicates on the nodes in the prefix path, including the separate node; and *the inner predicates* – the predicates on the nodes in the suffix path. The sub-tree rooted at the separate node is called the *inner predicate sub-tree*. *Outer predicate sub-tree* is the query pattern tree subtracts the inner predicate sub-tree. The outer predicate sub tree in query is less restrictive than that of the XML view.

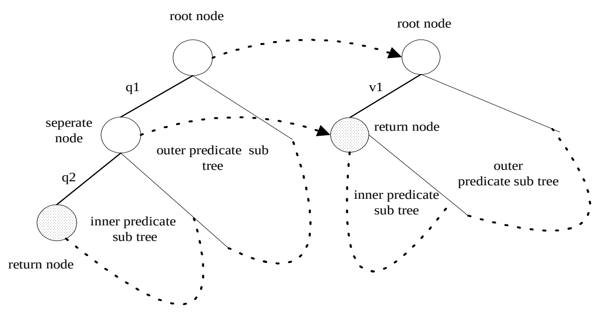

Fig. 2: Mapping from the query definition to view

Due to the different roles of the single XPath in the main path and the predicate sub tree, we build two XML view management auxiliary structures: MPTree and PPLattice to organize the XML in a more manageable way. MPTree is constructed from the main path of the XML views set to generate the query rewriting candidate plans. PPLattice is constructed from the predicates sub tree to validate the query rewriting candidate plan. Both MPTree and PPLattice can reduce the search space in the query rewriting plans over the multiple views. The framework of our method can be illustrated in the following figures:

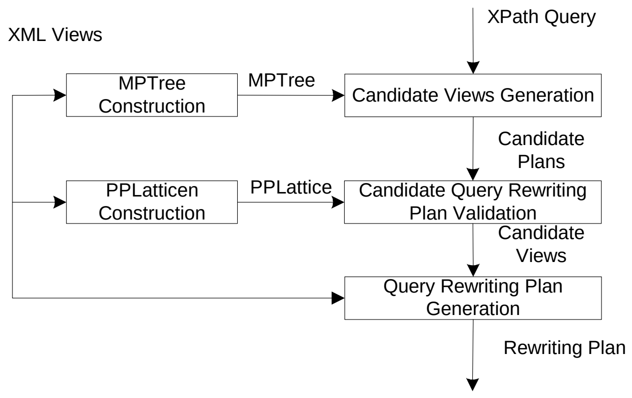

Fig. 3: the framework of our method.

From Fig.3, we generate the candidate query rewriting plans from the MPTree which is constructed from the main path of the XML view for a given XPath query. We validate the candidate query rewriting plans with the PPLattice at the first stage. If the candidate plans pass the validation at the first stage with the PPLattice, we validate the query rewriting plan as a whole tree at the next stage and generate the final query rewriting plan.

3.2. MPTree

MPTree contains the information of main paths from all existing views. Intuitionally, MPTree can be generated via the merging the main path of all XPath views in a top down fashion.

Definition 6: The MPTree T=(V,E,r) can be constructed from XPaths set P={p_1,..p_n}. For each node n in XPath $p_i \in \{p_1,..p_n\}$, if there is one node n_1 in XPath $p_j \in \{p_1,..p_n\}$ $i \neq j$, the path from the root node in p_i to node n is the same as the path from the root node in p_j to node n_1, we merge n_1 and n together. r is the root node of any XPath p. We also annotate the return node of XPath p_i with the related id i.

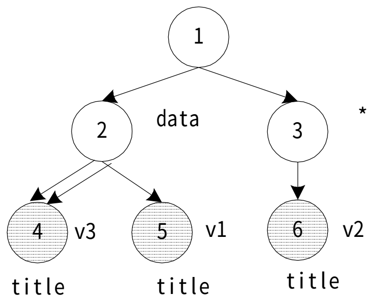

Fig. 4: the sample of MPTree

From Fig.4, we notice that sub path /*data* is shared by v_1 and v_3. If we generate the query rewriting plan against the XML views on MPTree, the sub path /*data* can be considered only once.

Two main operations are related with the MPTree. The first one is the construction of MPTree. This operation can be implemented in the same way as the definition of MPTree, which can be implemented in linear time in the size of the nodes of the main path of the XPath views. The second one is to generate the candidate view rewriting plan from the MPTree. The operation can be handled in the following way:

Given an XPath query q with the single XPath tree T_q=(V_q,E_q,r_q) and the MPTree MT=(V_v,E_v,r_v) which is constructed from the main paths of all views, we construct the *composite tree* R=(V_p,E_p,r_p) over MT and T_q by the following rules: (1) the root node r_p of R can be initialized as the r_p=[r_q,r_v]. (2) Given the node n_1=[n_{q1},n_{v1}]$\in V_p$, (n_{q1}, n_{q2})$\in E_q$, (n_{v1}, n_{v2})$\in E_v$,

	Type(n_{q1},n_{q2})= "PC"	Type(n_{q1},n_{q2})= "AD"
Type(n_{v1}, n_{v2})= "PC"	n_2=[n_{q2},n_{v2}] is created with type(n_1,n_2)="PC" if the element of n_{q2} and n_{v2} are the same	n_2=[n_{q2},n_{v1}] is created with the type (n_1,n_2)="PC" and n_3 = [n_{q2},n_{v2}] is created with the type(n_1,n_3)="AD" if the element of n_{q2} and n_{v2} are the same,
Type(n_{v1}, n_{v2})= "AD"		n_2=[n_{q2},n_{v1}] is created with the type (n_1,n_2)="PC" and n_3 = [n_{q2},n_{v2}] is created with the type(n_1,n_3)="AD" if the element of n_{q2} and n_{v2} are the same,

Table 1: the Composition Rules

For each main paths of XML view v in the MPTree, we check whether there is one node (n_1, n_2) in the composite tree R, where n_2 is the return node for v, we know that n_1 is a separate node in query main path q for v. In other words, the suffix path with separate node n_1 in q can be used as the query q rewriting plan for view v. From the composition rules, we know that the suffix path in the query is less restrictive that the main path of the view.

3.3. The PPLattice

We further check the validation of the candidate query rewriting plan by establishing the containment mapping from the query outer predicate tree to the view outer predicate tree. From the property of the containment mapping, we can find a mapped single XPath in the view for each single XPath in the query if one valid mapping exists for the whole tree pattern. In other words, if we can not find a mapped single XPath for the query tree, the view will not be contained in query. In order to speed up the validation of the candidate query rewriting plan and reduce the cost of the validation test for the whole tree, we can detect the restrictiveness between the single XPath in XPath view and single XPath in XPath query at the first stage. If it is not satisfied, the view will not be considered further in the following step.

The mapping from the single path p_1 in the query to the single path p_2 to the view indicates that p_1 is more restrictive than p_2. From the existing study [9], the containment test can be implemented in a polynomial time when the single XPath support {//,/,*}. We can exploit such a feature to speed up the query containment for the single XPath.

In order to detect whether there are single XPaths in XPath views more restrictive than single XPaths in XPath query efficiently, we propose the structure of PPLattice:

Definition 7: PPLattice G=(V,E,T,B) can be constructed from XPaths set P={p_1,..p_n}. Each node $n \in V$ represents the different single XPath p in each XPath in P. The single XPath expression $n_{[S]} \in S$ is annotated on the node n. For node $n_1 \in V$ and node $n_2 \in V$, if there is no node $n_3 \in V$, where n_1 is more restrictive than n_2, n_1 is more restrictive than n_3, n_3 is more restrictive than n_2, we build an edge $e \in E$ from node n_2 to n_1. The *top nodes set T* include the nodes without the parent node. The *bottom nodes set B* includes the nodes without the child nodes.

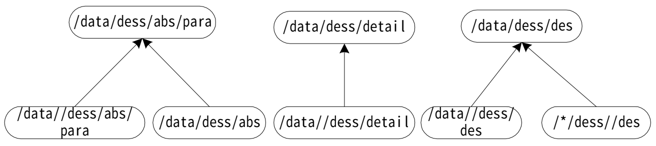

Fig. 5: the sample of the PPLattice

As for a node *n* in PPLattice, there maybe several parent nodes. Therefore, the PPLattice does not take the form of tree structure. In addition, we notice that there is no nodes sequence (n_1, n_2), (n_2, n_3), (n_1, n_3) in PPLattice. Since we can infer that the path for n_3 is more restrictive than that of n_1 from other links transitivity, we need not construct link from n_1 to n_3. Therefore, the edge in PPLattice can be reduced.

In addition, there is no nodes sequence cycle in the PPLattice. We can prove such property in the following: Without loss of the generality, suppose that there are links (n_1, n_2), (n_2, n_3), (n_3, n_1) in PPLattice, we know that the path for n_1 is more restrictive than that for n_3 and the path for n_3 is more restrictive than that for n_1. In this way, n_1 and n_3 are equivalent. However, the nodes in PPLattice are different.

There are three main operations on the PPLattice, including the construction of the PPLattice, the pruning of the PPLattice based on view ID, the location of nodes with path more restrictive than a given single XPath.

The PPLattice is constructed from the predicate sub trees for each view in the views set. The PPLattice is set empty initially. Given single XPaths set *N* for the predicate sub tree, we insert single XPath into the existing PPLattice. We set the current top layer *T* as the top nodes set. For a node *t* constrcuted from the single XPath in *N*, we detect the relationship between node *t* and node *p* in *T*. If there is no containment relationship between *p* and t, we just remove *p* from the nodes set *T*. If *t* is more restrictive than that of *p*, we build edge between the *p* to *t* directly. In other case, we check the restrictiveness between children nodes of *p* and *t*. If there is one child node *pc* of *p* with more strong restrictiveness than that of *t*, we remove node *p* from top set *T*, and we add child node *pc* into *T*. We apply the above rules untile there is no changes of the Top set *T*. The final nodes set T denotes the direct parent nodes set for node *t*.

We handle the lower layer nodes set *B* similarly. The final nodes set *B* represents the direct nodes set for node *t*. In this way, the node *t* can be inserted into the PPLattice and the link between each node in T or B to node t is established.

We also need to prune the PPLattice since only part of the views are involved in the candidate query rewriting plan while PPLattice is constructed from all XML views. For each XPath view *v* in the view set, we locate the node *n* for the single path in *v* in the PPLattice and annotate node *n* ‘Usefull’. We remove the nodes without annotation and reconstruct the link based on the original relationship between nodes.

With the pruned PPLattice, we try to locate the node with the more restrictive single path than the given single path *p* in query. We set the current top nodes *T* as the top nodes set, and determine the restrictiveness of the each node *t* of *T* and single path *p*. If there is no containment relationship between *t* and *p*, we just remove *t* from *T*. If the path for *p* is more restrictive the path for node *t*, we remove *t* from *T* and need not consider any child or descendant node of *n* in the following. If the path for *t* is more restrictive than that of *p*, we add all children nodes of *t* into *T* and apply the rules on *T* iteratively. *T* contains the nodes with path more restrictive than path *p* finally.

4. The Query Rewriting Plan Generation Based on MPTree and PPLattice

With the MPTree and PPLattice, we try to build the query rewriting plan efficiently. The whole process can be illustrated in the following figure.

Given one XPath, we generate the related tree pattern. We decompose the tree pattern into the main path p_{m} and a set of predicate single path p_{p}. we check the candidate views set V_{pmi} on MPTree for p_{m}, and generate the query rewriting plans over the multiple views.

We obtain the candidate views ID from V_{pmi}. We reconstruct the PPLattice based on the views ID. The following operation will be operated on the newly built PPLattice. We locate the nodes with the more restrictive path than satisfied single path node which is more restrictiveness than p_{p}. We obtain the views set for each p_{p} and make the interaction among them. The view in the intersection results passes the validation with the MPTree and PPLattice.

We need to provide the sound query containment test method to validate the candidate query rewriting plan which has meet the requirement of MPTree and PPLattice. The containment operation can be implemented with the following containment operation.

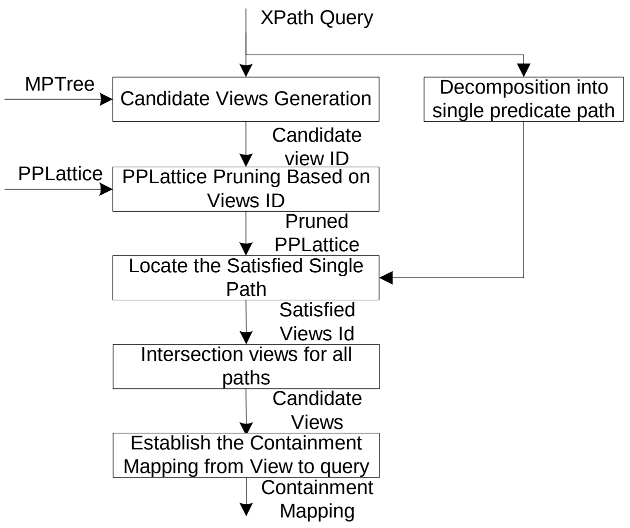
XPath Query
MPTree
Candidate Views Generation
Decomposition into single predicate path
Candidate view ID
PPLattice
PPLattice Pruning Based on Views ID
Pruned PPLattice
Locate the Satisfied Single Path
Satisfied Views Id
Intersection views for all paths
Candidate Views
Establish the Containment Mapping from View to query
Containment Mapping

Fig. 6: The XPath Query Rewriting Plan Based on MPTree and PPLattice.

Definition 8: (The containment of the tree pattern) Given an outer predicate sub tree $T_q=(V_q,E_q,r_q)$ and an outer predicate sub tree $T_v=(V_v,E_v,r_v)$, we construct the tree pattern $P=(V_p,E_p,r_p)$ by the following rules:

1) The nodes in the P are initialized with the nodes set in the form of $[n_{q1},n_{v1}]$, where n_{v1} is the leaf node n_{v1} in XPath view, node n_{q1} is the node with the same element as that of the or the wildcards.
2) The node $[n_{q2}, n_{v2}]$ is added into V_p where $n_{q2}\in V_q$ and $n_{v2}\in V_v$, for each node $[n_{q3}, n_{v3}]$, if the single path from n_{q2} to n_{q3} in T_q is less restrictive than the single path from n_{v2} to n_{v4} in T_v. As for the node n_{q2}, we called that n_{q2} find the mapped node in T_v.
3) If all the branching nodes T_q find the mapped node in T_v, and there is one node $[n_{q1}, n_{v1}]$ where $n_{q1}=r_q$ and $n_{v1}=r_v$, the containment between T_q and T_v returns *TRUE*, else returns *FALSE*.

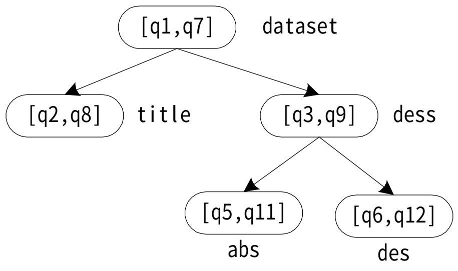

Fig. 7: The illustration of the Containment Mapping

Basically, the result tree of containment operator encodes the containment relationship between the XPath view to the XPath query in the newly created node. Taking the outer predicate sub tree for q_1 and v_1 in Fig.1 as example, the initial nodes of the containment operation result pattern contains $\{[q_5, q_{11}], [q_6, q_{12}]\}$ in Fig.8 since both q_5 and q_{11} are labeled with element *abs*, both q_6 and q_{12} are annotated with the element *des*. We know that $[q_3, q_9]$ will be new nodes in T_R considering the requirement. We take the similar rules until we find that one node $[q_1, q_7]$ will be new state in T_R. The mapping from query q_1 to view v_1 is encoded in the newly generated node.

5. Related Work

The former related work includes view-based querying on the relational model, view-based query answering on the semi-structured data, cache management in the XML query process, and XPath optimization with schema.

Answering query using views has been extensively studied on the relational model [6,7,8]. Two fundamental algorithms on relational data, bucket and inverse rule algorithms, have been proposed in [6,7]. It is not a trivial work to extend the result on the relational model to the nested data model due to the semantic mismatch among two models and different expressive power of two query languages.

Some attempts have been made on the semi-structured data model or the graph data model. The method supports [11] the nested query expression and the results reconstruction, however, it does not support features similar to "//" or "*" in the XPath. Regular path query rewriting discussed in [4] on graph model supports the regular expressions. However, it does not handle the result reconstruction.

XPath rewriting at server side is studied in [1], which handles the XPath rewriting over XPath views if we regard the XML index as a special case of XML view. It extends the query containment test [11] and proposes an incomplete but efficient XPath{//,/,[],*,|} rewriting algorithm. However, this work does not take the multiple XML views into account. XPath rewriting at client side is studied in [2]. It provides the polynomial method to handle the XPath query rewriting when the features of the XPath are limited. The XML views selection from the multiple XML queries is studied in [3].

6. Conclusion and Future Work

In this paper, we proposed MPTree and PPLattice to manage the multiple XML views, and develop related algorithm to generate the query rewriting plans efficiently.

Reference

[1] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, H. Pirahesh: A Framework for Using Materialized XPath Views in XML Query Processing. In Proc of

VLDB 2004. pp60-71.

[2] W.H.Xu, Z. M.Özsoyoglu. Rewriting XPath Queries Using Materialized Views. In Proc. of VLDB 2005, pp 121-132.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. (Working Draft). Available at http://www.w3c.org/TR/xquery.

[4] D. Calvanese, G.D.Giacomo, M. Lenzerini, M. Vardi. Rewriting of regular expressions and regular path queries. In Proc of POD 1999, pp 194-204.

[5] J. Clark. XML Path language(XPath), 1999. Available at the W3C, http://www.w3.org/TR/XPath.

[6] M.R.Genesereth, A.M. Keller, Oliver M. Duschka: Infomaster: An Information Integration System. In Proc of SIGMOD 1997, pp 539-542.

[7] G.Grahne, A.O.Mendelzon. Tableau techniques for querying information sources through global schemas. In Proc of ICDE 1999, pp 32-347.

[8] A.Halevy: Answering queries using views: A survey. In VLDB Journal. 10, 4 (2001), pp270-294.

[9] G.Miklau, D.suciu. Containment and equivalence for an XPath fragment. In Proc of PODS 2002, pp 65-76.

[10] F.Neven, T.Schwentick: XPath Containment in the Presence of Disjunction, DTDs, and Variables. In Proc of ICDT 2003, pp 315-329.

[11] Y.Papakonstantinou, V.Vassalos. Query rewriting for semi-structured data. In Proc of SIGMOD 1999, pp 455-466.

[12] NASA data set. Available at http://www.cs.washington.edu/research/xmldatasets/www/repository.html#nasa.