A Framework for Using Materialized XPath Views in XML Query Processing

Andrey Balmin Fatma Özcan Kevin S. Beyer Roberta J. Cochrane
Hamid Pirahesh

IBM Almaden Research Center, San Jose CA 95120
{abalmin, fozcan, kbeyer, bobbiec, pirahesh}@us.ibm.com

Abstract

XML languages, such as XQuery, XSLT and SQL/XML, employ XPath as the search and extraction language. XPath expressions often define complicated navigation, resulting in expensive query processing, especially when executed over large collections of documents. In this paper, we propose a framework for exploiting materialized XPath views to expedite processing of XML queries. We explore a class of materialized XPath views, which may contain XML fragments, typed data values, full paths, node references or any combination thereof. We develop an XPath matching algorithm to determine when such views can be used to answer a user query containing XPath expressions. We use the match information to identify the portion of an XPath expression in the user query which is not covered by the XPath view. Finally, we construct, possibly multiple, compensation expressions which need to be applied to the view to produce the query result. Experimental evaluation, using our prototype implementation, shows that the matching algorithm is very efficient and usually accounts for a small fraction of the total query compilation time.

1 Introduction

With large amounts of data represented and exchanged as XML documents, there is a pressing need to persistently store and efficiently query large XML collections. To address this requirement, W3C has proposed an XML query language, *XQuery* [17]. At the same time, ANSI and ISO has defined SQL/XML [1] (XML-Related Specifications), a new part of SQL standard which extends SQL to handle XML data.

XPath [16] is the W3C recommendation for navigating XML documents, which is designed to be embedded in a host language such as XQuery, XSLT or SQL/XML[1]. It is a reference based language; hence, subsequent expressions on the results an XPath expression may traverse the document in both forward and reverse directions.

XPath expressions often define complicated navigation, resulting in expensive query processing, especially when executed over large collections of documents. As a result, optimization of XPath expressions is vital to efficiently process XML queries.

In this paper we propose a framework for exploiting materialized XPath views to expedite processing of XML queries. The views may contain copies of XML fragments, or node references which point into the original document. The framework also considers value and structure indexes built on top of the views to efficiently locate information within them. We do not propose any particular structure to assist query processing, but rather a general framework for exploiting a large class of such structures.

Our framework addresses rewriting of XPath expressions that are embedded in XQuery. Consider the following XQuery:

Q1 : $for\ \$i\ in\ collection('URI')//order$
$where\ \$i//@price > 100$
$return$ <order> {$\$i//lineitem$} </order>

This query contains three XPath expressions, and all three return different types of values. In the *for binding*, the XPath expression //order returns node references to *order* nodes. In the *where clause*, the comparison \$i//@price > 10 requires the typed value

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

[1]The latest draft of SQL/XML embeds XQuery which includes XPath

of the price attributes, and finally within the element constructor copied subtrees rooted at *lineitem* nodes are needed.

As the above example illustrates, it is worthwhile to store additional "values" with the results of XPath expressions. In this paper, we explore XPath views containing XML fragments, typed data values, node references, full paths and a combination thereof.

The problem of rewriting queries using materialized views becomes even more relevant in the world of XML query processing, since XML indexes also can be modeled as materialized views. Indeed, most of the recently proposed XML indexing schemes (e.g., [14, 9, 3, 5, 11, 7]), can be viewed as materialized views that contain a value and a node reference for every element in the collection. Similarly relational indexes can be viewed as very limited materialized views that contain column values and row IDs. The materialized view model is especially accurate for *partial* XML indexes, which contain only nodes that satisfy a certain XPath expression. An index does not have to contain all elements in the collection. After all, relational indexes are rarely defined for all columns of all tables in the database. A partial XML index is likely to improve access time and reduce index maintenance costs. We envision such indexes to be very useful for XML query processing.

Consider a query $Q = collection('uri1')$ $//order[date > "Jan\ 1, 2004"\ and\ price > 100]$, which asks for recent expensive orders. First, let us consider a view $V_1 = collection('uri1')//*$ which contains data values and node references of all elements in a collection. We model this view as a table with attributes *value* and *reference*. One way to execute the query using this view is to find all elements with values greater than 100, then follow the node references to the original documents and select only *price* elements that have *order* parent with $date > "Jan 1, 2004"$.

Now let's assume that, in addition to values and node references, V_1 also contains elements' full path, i.e., a list of all ancestor tags. This enables us to specifically locate *price* elements with *order* parents and then follow node references to execute the expression $..[date > "Jan1, 2004"]$.

Of course not all materialized XPath views can be used to answer a query. For example, it is not beneficial to use the view $V_2 = collection('uri1')//order[date > "Feb\ 1, 2004"]/price$, since it does not contain January orders. The document collection needs to be scanned anyway to locate January orders.

The above examples illustrate two main problems associated with answering XML queries using materialized XPath views. First, an XPath query containment is required to make sure that a view can be used to answer a query. Second, a *compensation* expression needs to be constructed, that would compute the query result using the information available from the view.

We address the XPath query containment problem with an XPath matching algorithm. The containment problem was shown to be co-NP complete for a restricted subset of XPath by [10]. We propose an efficient polynomial-time matching algorithm which is sound and works in most practical cases.

The algorithm is based on the observation made in [10] that a total node mapping from view nodes to query nodes implies containment for conjunctive XPath expressions. We build on the same observation, but extend it to a more functional subset of XPath that includes value predicates, disjunction and the six axes allowed in XQuery.

The mappings produced by the algorithm are used to construct the compensation expression. A mapping between view and query trees is not necessarily unique. Multiple mappings imply that the view can answer different predicates and/or fragments of the query. In those cases the compensation needs to combine information from multiple instances of the view.

To enable the best compensation expression, the matching algorithm produces all possible mappings. The number of tree mappings may be exponential, but we are able to encode them in a polynomial size structure. The running time of the matching algorithm is also polynomial.

The rest of the paper is organized as follows. In Section 2 we describe the XPath materialized views supported by our framework. We present our XPath query matching algorithm in Section 3. In Section 4 we discuss the framework that uses the matching algorithm to decide when and how to use the materialized views to answer a query. We present experimental evaluation of the matching algorithm in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Materialized XPath Views

In relational databases, indexes and materialized views are two well-known techniques to accelerate processing of expensive SQL queries. In this section, we explore a class of materialized XPath views to speed-up processing of XQuery or SQL/XML queries. We consider XPath views containing XML fragments, typed data values, node references, full paths and a combination thereof.

One class of XPath views may contain only typed data values of nodes. Storing typed values facilitates computation of value based comparison predicates. B^+-tree indexes on typed values further expedites query processing. For example, if an XPath view contains the typed values of $/lineitem/@price$, then it can be used to answer the comparison predicate $/lineitem/@price > 1000$.

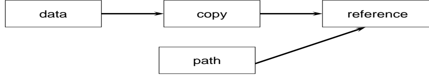

Figure 1: Extraction Type Hierarchy

Note that this type of XPath views can also be considered as a regular value index. However, there is one important restriction of such XPath views: The XPath expression in the comparison has to be exactly the same as the view XPath expression. Because the XPath view does not contain enough information if the query XPath expression does not exactly match. Consider the query $/lineitem/@price < 100$. An XPath view $V = //lineitem/@price$ cannot be used to evaluate this comparison predicate, because there is no way to determine from the view that lineitem nodes are top-level element nodes.

To remedy this problem, one may also store full paths in an XPath view. A full path of a node is defined as a list of ancestor tags. Such paths can be computed when XML data is processed and inserted into the view results. These full paths can be very useful if the defining XPath expression of the view contains descendant axes and/or wildcard name tests. For example, we can use the XPath view $V = //@*$ to evaluate comparison predicates $/lineitem/@price > 100$, /lineitem/order/@quantity < 10, $/lineitem/*/@amount = 1000$, and etc.

Storing typed values is only helpful to evaluate comparison predicates, whether in where clause of a FLWOR expression, or in a predicate of an XPath step expression. However as shown in **Q1** in the Introduction, XPath expressions are also used in other contexts. To evaluate those queries, it is often required to apply further navigation to the results of an XPath view. In such scenarios, we need to store node references in the materialized XPath view.

Consider the query **Q1** and the XPath view $V = //order$. If V only contains typed values of orders, we cannot use it to process **Q1**. However, if V contains references to order nodes, then we can use this view, as we can execute the XPath expressions $i//@price and $i//lineitem, by using the results of V.

Sometimes it is also beneficial to store actual copies of XML fragments in an XPath view. For example, it might be sufficient to store copies to answer value-based expressions of SQL/XML. Furthermore, concurrency control is easier in the case of copy semantics. Note that copy semantics loses the parent property as well as node identity. This implies that XPath views containing copies can only be used to answer XQueries when subsequent operations on the results of the view do not require to navigate to the parent or ancestors, or require node identity, such as node comparisons, and sequence operations of XQuery.

One can think of different kinds of materialized XPath views which contain a combination of node references, full paths, typed data values and copied XML fragments. We refer to node references simply as *references*, typed data values as *data*, copied XML fragments as *copy* and full paths simply as *paths* in the rest of this paper. A view definition language to create these different kinds of materialized XPath views is beyond the scope of this paper. Here, we assume that views are relations with the schema *(reference, copy, data, path)* and are defined with XPath expressions where view extraction point is marked with copy, data, path and reference extraction types. Figure 1 summarizes the relationships between these extraction types in terms of the information they represent. We explain how these different extraction points are used in Section 4.

3 XPath Matching Algorithm

In this section, we present an algorithm to decide if a given XPath view can be utilized in a user query. The algorithm finds tree mappings between the view and the query expression trees, and records them in a match structure. If a mapping exists then the view can potentially be used to evaluate the XPath expression in the user query. However, further computation may be necessary. We use the match structure produced by the algorithm to derive the *compensation*, which is an expression applied to the contents of the view to compute the query result.

In the remainder of this section, we first introduce our XPath representation. We then describe the basic algorithm, which concentrates on "structure-only" XPath queries, followed by an extension to handle comparison predicates.

3.1 XPath Representation

We represent XPath expressions as labeled binary trees, called *XPS trees*. An XPS node is labeled with its *axis* and *test*, where *axis* is either the special "root", or one of the 6 axes allowed in XQuery [17]: "child", "descendant", "self", "attribute", "descendant-or-self", or "parent". The *test* is either a name test, a wildcard test, or a kind test, such as node() or text(). The first child of an XPS node is called *predicate*, and it can be a conjunction (`and`), a disjunction (`or`), a comparison operator ($<, \leq, \geq, >, =, \neq, eq, ne, lt, le, gt, ge$), a constant, or an XPath Step (XPS) node. The second child, called *next*, points to the next step, and is always an XPS node. *Next* and *predicate* are optional. We use "null" nodes to denote missing children. An XPS node which does not have a next step, and is reachable from the root of the XPS tree by visiting only *next* children, is called the *extraction point*, since this node represents the result of the XPath expression. Example XPS trees can be found in Figure 2.

3.2 Basic Matching Algorithm

The algorithm described in this section traverses both the view and the query expression trees and computes

1	$matchStep(v, q)$	
1.1	if $(q = q_1 \wedge q_2)$	$matchStep(v, q) \rightarrow matchStep(v, q_1) \vee matchStep(v, q_2)$
1.2	else if $(q = q_1 \vee q_2)$	$matchStep(v, q) \rightarrow matchStep(v, q_1) \wedge matchStep(v, q_2)$
1.3	else if (v_{axis} = "descendant")	$matchStep(v, q) \rightarrow \bigvee \{matchChildren(v, c)\}, \forall c \in \{$ preorder traversal of $q\}$, such that $c_{axis} \neq$ "attribute", unless v_{test} = node()
1.4	else if $(v_{axis} = q_{axis})$	$matchStep(v, q) \rightarrow matchChildren(v, q)$
1.5	else	$matchStep(v, q) \rightarrow False$
2	$matchChildren(v, q)$	
2.1	if (v_{test} = "*") $\vee$ $(v_{test}{=}q_{test})$	$matchChildren(v, q) \rightarrow matchPred(v_{pred}, q) \wedge matchNext(v_{next}, q))$
2.2	else	$matchChildren(v, q) \rightarrow False$
3	$matchPred(v_{pred}, q)$	
3.1	if (v_{pred} = null)	$matchPred(v_{pred}, q) \rightarrow True$
3.2	else if (q = null)	$matchPred(v_{pred}, q) \rightarrow False$
3.3	else if $(v_{pred} = v_1 \wedge v_2)$	$matchPred(v_{pred}, q) \rightarrow matchPred(v_1, q) \wedge matchPred(v_2, q)$
3.4	else if $(v_{pred} = v_1 \vee v_2)$	$matchPred(v_{pred}, q) \rightarrow matchPred(v_1, q) \vee matchPred(v_2, q)$
3.5	else	$matchPred(v_{pred}, q) \rightarrow matchStep(v_{pred}, q_{pred}) \vee matchStep(v_{pred}, q_{next})$
4	$matchNext(v_{next}, q)$	
4.1	if (v_{next} = null)	$matchNext(v_{next}, q) \rightarrow True$
4.2	else if (q = null)	$matchNext(v_{next}, q) \rightarrow False$
4.3	else	$matchNext(v_{next}, q) \rightarrow matchStep(v_{next}, q_{pred}) \vee matchStep(v_{next}, q_{next})$

Table 1: Rules for finding containment mappings between expression trees

all possible mappings from XPS nodes of the view to XPS nodes of the query expression, in a single top-down pass of the view tree. For the ease of readability, we denote the XPath expression defining the view with V, and the XPath expression in the user query with Q.

In the basic algorithm we restrict the view and query XPS trees to contain only AND, OR nodes and XPS nodes with child, attribute, or descendant axis.

Table 1 summarizes the basic algorithm in terms of the four functions used. Every function of the table evaluates to Boolean. The algorithm is invoked by the initial call $matchStep(v_root, q_root)$, and there exists a match if this call evaluates to *true*. The first rule whose condition is satisfied is fired for each function.

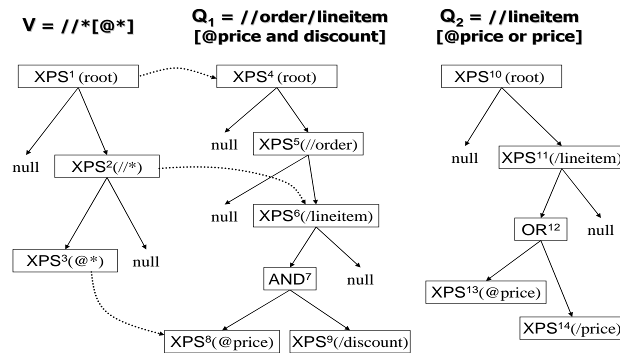

Figure 2: Example view definition, matching and non-matching query expressions

Rule 1.1 handles the situation where the query expression can be more restrictive than the view definition. It is sufficient for one of the conjuncts of q_{pred} to be mapped by a node of V. For example, the view $V = // * [@*]$, which contains all XML element nodes which have an attribute, can be used to evaluate $Q_1 = //order/lineitem[@price\ and\ discount]$ as shown in Figure 2. Dotted lines denote the mapping.

Rule 1.2 says that if one disjunct of q_{pred} is mapped by a node $v \in V$, then v also has to map to some node in the other disjunct of Q. For example, the same V of Figure 2 cannot be used to evaluate the expression $Q_2 = //order/lineitem[@price\ or\ price]$, which asks for `lineitem` nodes, which have either a `price` attribute or a `price` element.

When the view node contains a "descendant" axis, we need to keep looking for matches down in the tree, even if the current query expression node matches (rules 1.3). For example, in Figure 2, we will try to map $XPS^2(//*)$ to XPS^5 (//order), XPS^6(/lineitem), and XPS^9(/discount). We do not include rules that handle "self" and "descendent-or-self" due to lack of space. "Parent" axis is handled by rewriting the expression into an equivalent XPS tree that uses forward axes only. The rewriting is done using transformations similar to the ones proposed in [13]. If the axes match, we try to match the predicate and next children of the view node (rule 1.4). If there is no match (rule 1.5), the algorithm returns *false*.

When matching children, if the tests match, then we try to match the predicate and the next step of v (rule 2.1). If v does not have a predicate then the step trivially matches (rule 3.1). Recall that the view expression can not be more restrictive than the query. Hence, if v has a predicate and q does not, then the match fails (rule 3.2). The next children of XPS nodes are matched in the same fashion.

The rule 3.3 states that if there is a conjunction

in V, then both conjuncts has to map to some node in Q. However, it is sufficient for one disjunct in V to participate in the mapping (rule 3.4). For example, the view $V = //order[@price\ or\ lineitem/@price]$ can be used to evaluate Q_1 of Figure 2.

The rules of $matchNext()$ are similar to those of $matchPred()$.

As the predicate of an XPath step may contain a nested XPath expression, we try to match v_{pred} both to q_{pred} and q_{next} (rule 3.5), and match v_{next} to both q_{pred} and q_{next} (rule 4.3). For example, a view expression $V = //a[b/c]$ matches the XPath expression $Q = //a/b[c]$.

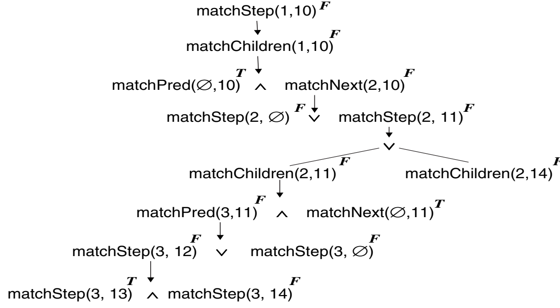

Figure 3: Execution of the matching algorithm for V and Q_2 of Figure 2

Figure 3 shows the execution of the algorithm as the rules fire, and return *false*. There is no match because there is no XPS node in the view that maps to $XPS^{14}(/price)$. That is, the view does not contain `lineitem` elements with a `price` sub-element, but no attribute.

3.3 Recording the Match

We need to preserve information about all tree mappings found by the algorithm to fully take advantage of data stored in the view, when constructing the compensation expression. We keep track of all mappings in a *match matrix* structure, which also facilitates matching intradocument joins (discussed in Section 3.5), and reduces time complexity of the algorithm by eliminating redundant computation.

The basic matching algorithm may generate exponential (in the number of XPS nodes) number of tree mappings. For example, consider a view that consists of n nodes $//a//a \ldots //a$ and a query expression that consists of m nodes $/a/a \ldots /a$, where $m > n$. Any view node v can map to any query expression node q such that v's parent maps to some ancestor of q. Thus for any subset of n query nodes, there is exactly one mapping from the nodes of the view to these query expression nodes. Hence, there are C_n^m distinct tree mappings of the view to the query expression. However, a lot of information in these mappings is redundant, since the space of mapping options for a node v depends only on the mapping of v's parent (which we will refer to as *mapping context*) and is independent of all other view node mappings. In other words, *matchStep*() function of our algorithm may be called multiple times with the same parameters v and q, which is wasteful, since it is guaranteed to return the same result, due to the top-down nature of the algorithm.

Match matrix allows us to encode an exponential number of tree mappings in a polynomial size structure, by recording all possible contexts for each node mapping. It also reduces running time of the algorithm to polynomial, by eliminating redundant computation.

Intuitively, each cell of the match matrix corresponds to a pair of view and query XPS tree nodes. Result of each *matchStep*() call is cached in the corresponding cell of the matrix. Before executing each *matchStep*() the matrix is checked, and if the same pair of nodes has already been matched, we return the result stored in the matrix instead of running the *matchStep*() again. More precisely, each row of the matrix corresponds to an XPS node of the view tree, and each column corresponds to an XPS node of the query expression. Each cell of the matrix may contain one of three possible values: "empty", "true" or "false". All cells are initialized as "empty".

In addition to the cell values, we also record directed edges between cells to represent the context in which the mapping was detected for a pair of nodes. An edge $(i, j) \rightarrow (k, l)$ means that (a successful) $matchStep(v_k, q_l)$ was called (possibly through other functions) from $matchStep(v_i, q_j)$. Recall that matching proceeds in a top-down traversal of the view tree, which means that v_i is a guaranteed to be an ancestor of v_k. Thus the edges form a directed acyclic graph (DAG) of matrix cells.

The *matchStep*() function of the algorithm of Table 1 is modified as follows: Before executing a call $matchStep(v_i, q_j)$, cell (i, j) of the matrix is checked. If the cell is empty, the function is executed. Otherwise the function returns the content of the cell. After a call $matchStep(v_i, q_j)$ is executed and returns *true* or $false$, we store this value in the matrix cell (i, j). If the result was *true*, we also create the edge $(k, l) \rightarrow (i, j)$ to the structure, where $matchStep(v_k, q_l)$ is the first *matchStep*() function on the call stack. This edge signifies that node v_i matches q_j in the context of the match of v_k to q_l.

EXAMPLE 3.1 Consider a hierarchy of employees, where each employee element has salary and bonus attributes, and zero or more employee sub-elements. Consider a view that contains all attributes in a subtree of any employee, and an XPath expression that asks for the salary of employees who, together with their direct managers, have bonuses. The XPS trees for these view and query are shown in Figure 4. Note that according to XPath standard $//@*$ is translated into $/descendent-or-self :: node()/@*$. The result-

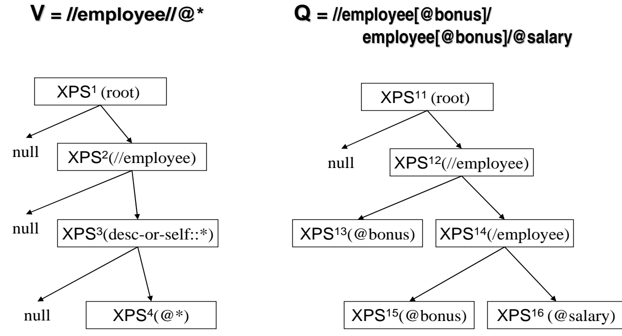

Figure 4: Example view and query expression trees.

V \ Q	XPS11 root	XPS12 //employee	XPS13 @bonus	XPS14 /employee	XPS15 @bonus	XPS16 @salary
XPS1 /root	T					
XPS2 //employee		T		T		
XPS3 dos::*		T		T		
XPS4 @*			T		T	T

Figure 5: Match matrix for view and query expression trees of Figure 4.

ing match matrix for these view and query expression is shown in Figure 5. For clarity we only show "true" cells.

The matching starts by calling $matchStep()$ on root nodes XPS^1 and XPS^{11}, which in turn tries to match $XPS^2(//employee)$ to all XPS nodes of the query[2]. Two of these match attempts will succeed. Let us consider them one at a time.

First, $matchStep(XPS^2, XPS^{12})$ calls $matchStep(XPS^3, XPS^{12})$, which in turn calls $matchStep(XPS^4, XPS^{13})$. The later returns $true$, which is recorded in cell (4,13) of the matrix. At this point the first edge $(3, 12) \rightarrow (4, 13)$ is added to the structure.
$matchStep(XPS^3, XPS^{12})$ also returns $true$, so we set $(3, 12) = true$ and add edge from (2,12) to (3,12). Similarly we find matches (3,14), (4,15) and add edges $(2, 12) \rightarrow (3, 14)$ and $(3, 14) \rightarrow (4, 15)$.

Now consider execution of the call $matchStep(XPS^2, XPS^{14})$. It starts by calling $matchStep(XPS^3, XPS^{14})$. However, cell (3,14) is not empty – it has already been set to $true$, so we immediately return $true$, without re-executing the call. We also add an edge from (2,14) to (3,14), since the same match has been found in a new context. Finally, we set $(3, 12) = true$ and add edge $(1, 11) \rightarrow (2, 14)$ to complete the structure.

Notice that this structure encodes five distinct mappings of the view tree into the query. □

3.4 Handling Comparison Predicates

In this section, we extend our matching algorithm to handle comparison predicates, i.e., expressions of the form L op R, where op is one of XQuery general or value comparison operators, and L and R are either an XPS node or a constant.

Needless to say, predicate conditions complicate the matching algorithm. For example, consider the view $V = //order/*[@price > 60]$ that matches the query expression $Q = //order[lineitem/@price > 100]$.

[2]For simplicity we omit intermediate $matchChildren()$, $matchPred()$, and $matchNext()$ calls in this example.

Their XPS trees are shown in the upper part of Figure 6. Notice that view XPS node $XPS^2(/*)$ should match query expression node $XPS^5(/lineitem)$. However, in the view tree ">" appears under $XPS^2(/*)$, while in the query ">" is above $XPS^5(/lineitem)$.

To enable matching in the presence of comparison nodes, we normalize the expression trees by extracting the predicate conditions from the expression trees as a pre-process of the matching algorithm. For example, we create the trees V' and Q' shown in the lower half of Figure 6.

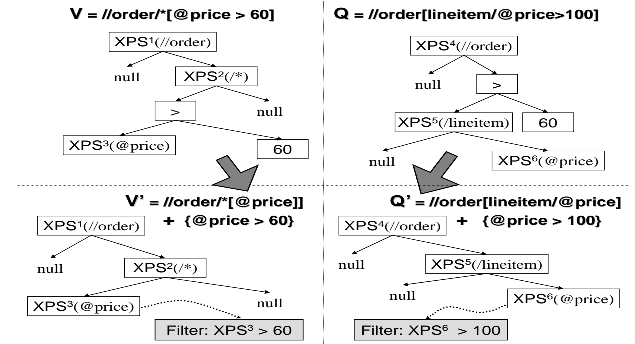

Figure 6: Normalizing the expression trees by extracting predicate conditions.

Both trees are traversed bottom-up and all comparison predicates are moved into a *filter list*. Each filter in the list is associated with the XPS node(s) it includes.

There are two types of filters: *local predicates* f_n of the form n op $const$, where n is an XPS node, and *intradocument join filters* f_{nm} of the form n op m, where both n and m are XPS nodes. During normalization, for each local predicate f_n, we replace its comparison operator node, with the subtree rooted at n, and associate f_n with n's extraction point n_e, which is an XPS node obtained by starting at n and following next children. For each intradocument join filter f_{nm}, we replace the comparison operator with an AND node, and associate f_{nm} with extraction points of both n and m.

EXAMPLE 3.2 Consider an XPath expression $//order[date = lineitem[@price > 100]/shipdate]$. It contains one local predicate and one intradocument join filter. Its normal form is $//order[date\ AND\ lineitem[@price]/shipdate]$ with two filters:

1. $price > 100$, associated with $@price$ XPS node
2. $date = shipdate$, associated with $/date$ and $/shipdate$ XPS nodes. $/shipdate$ is an extraction point, obtained by following the next step of the $/lineitem$ XPS node.

□

We extend the basic algorithm to match local predicates in the view during a single pass of the view tree. While the matching algorithm builds the mapping of view nodes to query expression nodes, it also checks whether for every local predicate f_v, that is associated with the current XPS node $v \in V$, there exists a predicate f_q associated with node $q \in Q$, such that v maps to q and f_q implies f_v ($f_q \rightarrow f_v$). Note that for local predicates implication can be detected in polynomial time, by examining the comparison operators and the constants in the predicates. Disjunction and conjunctions are handled as part of the matching algorithm.

EXAMPLE 3.3 The matching of V and Q of Figure 6 proceeds as follows. First, we extract the predicates, by replacing the $>$ nodes with their XPS children, producing trees V' and Q'. The extracted view predicate is associated with the XPS^3(@*price*) node of the view. The query expression predicate is connected to XPS^6(@*price*). Next, the algorithm of Table 1 proceeds by matching view node XPS^1 to query node XPS^4, which requires matching XPS^2 to XPS^5, which in turn attempts to match the view node XPS^3(@*price*), to the query expression node XPS^6(@*price*). At this point we find out that a view predicate $XPS^3 > 60$ is associated with node XPS^3, so we look for query expression predicates associated with the XPS^6 node. We find the predicate $XPS^6 > 100$, which is more restrictive than the view predicate. As a result, the algorithm matches XPS^3 to XPS^6. □

Notice that query expression predicates do not require any additional matching, since the query expression can be more restrictive than the view.

To match intradocument join filters in views requires matching both sides of the join predicate first. Hence, intradocument join filters are matched in a post processing step which is described next.

3.5 Matching Intradocument Joins

Recall that *intradocument join* filters represent joins that occur inside a single document. For example,

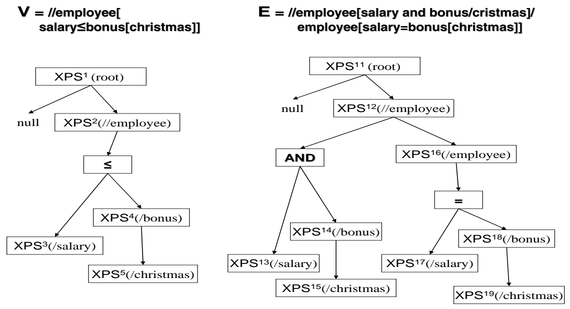

Figure 7: View and query expression trees with intradocument joins.

the expression $//employee[@bonus > @salary]/name$ contains an intradocument join filter $@bonus > @salary$. To match a view predicate f_v of the form $v_1\ op\ v_2$, we first need to find mappings for both XPS nodes v_1 and v_2. Given a match matrix that contains all node mappings for v_1 and v_2, we may need to prune some of these mappings if there is no query expression filter that implies f_v. In other words, the query has to contain a matching filter that is at least as restrictive.

Once we construct the match matrix, we analyze each intradocument join filter $f_v = v_1\ op\ v_2$ in the view.

If there is no query predicate $f_q = q_1\ op\ q_2$ that implies f_v, such that v_1 maps to q_1 and v_2 maps to q_2, we prune (i.e. set matrix cell to $false$) all node mappings that involve v_1 or v_2.

Finally, we clean-up the matrix, by repeating the following steps until no more modifications can be made.

- Remove all dangling edges for which either source or target matrix cell is not set to *true*.
- Remove orphan node matches, i.e., matrix cells with value *true* that do not have at least one incoming edge, are set to $false$.

We test that the resulting matrix is valid (i.e. encodes at least one tree mapping) by a single bottom up traversal of the view tree. An XPS node is valid, if its row in the matrix contains at least one *true* cell, and all its children are valid. An OR node is valid if at least one of its children is valid. An AND node is valid if all its children are valid. A matrix is valid, if the root of the view tree is valid.

EXAMPLE 3.4 Consider a view that lists all employees with Christmas bonuses no less than their salaries: $V = //employee[salary \leq bonus[christmas]]$, and an XPath expression that asks for employees that have salaries, Christmas bonuses and are managers of employees whose Christmas bonuses are equal to their salaries: $Q = //employee[salary\ and\ bonus/cristmas]/employee$

V \ E	XPS[11] root	XPS[12] //empl	XPS[13] /salary	XPS[14] /bonus	XPS[15] /cristms	XPS[16] /empl	XPS[13] /salary	XPS[14] /bonus	XPS[15] /cristms
XPS[1] /root	T								
XPS[2] //employee		T				T			
XPS[3] /salary			~~T~~				T		
XPS[4] /bonus				~~T~~				T	
XPS[4] /cristmas					T				T

Figure 8: Match matrix for XPS trees of Figure 7 and its pruning.

$[salary = bonus[christmas]]$. The XPS trees of these expressions are shown in Figure 7.

Intuitively, *employee* node in the view can map to either of the two *employee* nodes in the query. However, the first mapping ($XPS^2(//employee) \rightarrow XPS^{12}(//employee)$) is not valid because the intradocument join $[salary^3 \leq bonus^4]$ is not implied by any query filter (there is no filter involving $XPS^{13}(/salary)$ and $XPS^{14}(/bonus)$). Thus, node mappings $(XPS^3(/salary), XPS^{13}(/salary))$ and $(XPS^4(/bonus), XPS^{14}(/bonus))$ are pruned from the matrix. The node mappings $(XPS^2(//employee), XPS^{12}(//employee))$ and $(XPS^5(/cristmas), XPS^{15}(/cristmas))$ are removed from the matrix by the clean-up phase that eliminates dangling DAG edges.

The second mapping ($XPS^2(//employee) \rightarrow XPS^{12}(//employee)$) is valid, because the query predicate $bonus = salary$ is more restrictive than $salary \leq bonus$, and hence implies the view predicate.

Figure 8 shows match matrix for this example. Node mappings removed by pruning are crossed with an "X". Circled portion of the match is removed by the clean-up phase. □

3.6 Complexity of the Algorithm

Let us, first, consider space complexity of the algorithm. The size of the match matrix is $O(|V| * |Q|)$, where $|V|$ and $|Q|$ are the number of XPS nodes in the view and query expressions respectively. Each matrix cell can have at most $|Q|$ incoming edges (by construction an edge $(i, j) \rightarrow (l, k)$ may exist only if v_i is the parent of v_l). Thus the number of edges in the DAG is $O(|V| * |Q|^2)$.

The cost of constructing the matrix is also polynomial. The *matchStep* function has only $|V| * |Q|$ distinct sets of parameters. By definition of a match matrix, the same pair of nodes cannot be matched more than once. In the worst case (rule 1.3) a function call may expand into $|Q|$ function calls. Thus the algorithm runs in $O(|V| * |Q|^2)$ time.

The cost of pruning the matrix is a product of size of the matrix and the number of predicates extracted from V and Q, which is $O(|V|^2 * |Q|^3)$. Note that predicate subsumption ($f_q \rightarrow f_v$) can be checked in constant time, since f_q and f_v can only contain comparison operators: Disjunction or conjunction are handled by normalization and the matching algorithm, and no negation is allowed.

4 Matching Framework

The previous section defines matching of XPS view and query trees. In this section we exploit the resulting matches in a framework for rewriting XPath expressions using materialized XPath views.

When a view does not contain the exact results of an XPath query, we need to compensate, by applying some computation to the content of the view. This extra computation, called *compensation*, depends on what information is stored in the view.

Recall that a view expression extraction point is marked with one or more of four types: *reference*, *copy*, *path* and *data*. Thus a view is a relation with one attribute for each extraction type. To express compensation we use a variant of a relational algebra which consists of "select", "project" and "intersection" operators. These operators are extended to handle XML type. The select operator allows any XML comparison on *data* extraction type, any XPath expression on *reference* and *copy* extractions, and a new *match_path* operation, denoted $\sim$, on *path* extractions. We do not elaborate on the details of the algebra due to lack of space.

We construct compensation expressions by a two step process. First, we take a copy of the query XPS tree and *relax* it, i.e., eliminate conditions that are guaranteed to be satisfied given the view definition. For example, given $V = //a[@b]$ and $Q = //a[@b \wedge @c]$, the relaxed expression doesn't need to include a $[@b]$ predicate, since it is implied by the view. Second, we *optimize* the relaxed XPS tree and produce a compensation algebraic expression that fully utilizes information stored in the view. Since different extraction types may have the same information there may be multiple equivalent compensation expressions. We must chose amongst the alternatives.

In Sections 4.1 and 4.2 we describe compensation construction for a single mapping case. I.e. a single view whose extraction point maps to exactly one query node. We'll generalize the compensation for the case of multiple views and mappings in Section 4.3.

4.1 Eliminating unnecessary conditions

At this stage, a copy of the query XPS tree is *relaxed*, i.e. made less restrictive. We consider three possible types of XPS tree relaxation: removing a filter, replacing a name test with a *, and eliminating a step. The relaxed query produces the same result as the original query, when applied to the result of the view expression.

Notice that a relaxed query is always simpler than the original. We do not consider relaxations that complicate query processing, e.g. replacing child axis with descendant.

First, we identify a *compensation root node*, i.e., an XPS node in the query tree, that was mapped by the view extraction point. In this section we assume that the compensation root is unique. We will relax this assumption in Section 4.3.

The relaxation starts by constructing a Q' expression that is equivalent to the query Q, but starts at the compensation root. This is achieved by moving all XPS ancestors of the compensation root into it's predicate and reversing their axes. I.e. a "child" axis becomes "parent" and "descendant" axis is changed into "ancestor". For example, consider a query $Q = //a[b]/c[d]//e[f]/g$. If node e is the compensation root, we transform the query into:
$Q' = self :: e[f \wedge ancestor :: c[d \wedge parent :: a[b]]]/g$. If the compensation root is inside an intradocument join predicate, the new path expression starts with an upward traversal to the first XPS node outside of all such predicates, and then continues as if that node was the compensation root. For example, $Q = //a[b = c[d]]/e$, where d is the compensation root, translates into $Q' = self :: d/parent :: c/parent :: a[b = c[d]]/e$. This Q' start with upward traversal to the a node, which is the first XPS node outside the equality predicate.

While constructing the Q' expression we also transform the predicate of the new root node. All next steps inside the predicate are converted into an equivalent predicate step. E.g. $a[b[c]/d]$ is normalized into $a[b[c \wedge d]]$.

Next, we construct a V' expression that is equivalent to the view V, but starts at the view extraction point. This process is identical to Q' construction, further simplified by the fact that the extraction point, by definition, cannot occur in a predicate.

A "relaxed" query Q^r is obtained from Q' by the algorithm of Figure 9. The algorithm compares root node predicates of Q' and V' and eliminates all conjuncts of the Q' predicate, such that there is exactly the same conjunct in the predicate of V'. If the root of Q' or V' is not an AND node, we say that the entire predicate is a single conjunct.

4.2 Compensation Optimization

The result of query relaxation, Q^r, could be used as a compensation expression if $reference$ extraction type is available. However, following the reference to the data storage may be significantly more expensive than using data stored directly with the view. Thus, it is desirable to use other extraction types in the compensation expression. We decompose Q^r into one or more expressions, one per available extraction type, using the following rules.

1. If *data* extraction is available, it is used to answer any local predicate on the compensation root.

```
RelaxQuery(q,v)
copy Q^r = Q';
call RelaxQueryRec(Q^r.root,V'.root);
return Q^r;

RelaxQueryRec(q,v)
if (q.axis ≠ v.axis) ∨
  (q.axis = "descendant" ∧ q.name ≠ v.name)
    exit;
if (q.axis = "parent" ∧ q.name = v.name)
    set q.name = "*";
foreach q_c in conjuncts of q.pred do
  foreach v_c in conjuncts of v.pred do
    call RelaxQueryReq(q_c,v_c);
// If the recursive calls removed the entire predicate,
// and there is no next step, this step can be removed
if (q.pred==null ∧ q.next==null)
  remove q from Q^r;
```

Figure 9: Algorithm to eliminate unnecessary query conditions.

2. If *path* is available, it provides information about labels of ancestors of the compensation root.

3. If *copy* is available, use it if it can answer the portion of Q^r, not covered by *data* and *path*.

4. If $reference$ is available, it is used to answer all query conditions not covered by other extractions.

The compensation construction algorithm applies each rule in turn and marks nodes of the Q^r XPS tree that were covered by each extraction. Rules 3 and 4 construct an expression from all unmarked nodes in Q^r and execute it against the view or data storage respectively.

EXAMPLE 4.1 A view $V = //a$ with $copy, data$, and $path$ extractions can answer a query $Q = /b/a[. > 0]/c$ utilizing only information stored in the view, without accessing the data storage. The relaxed query in this case is $Q^r = self :: *[. > 0 \wedge parent :: b[parent :: root]]/c$. A $[. > 0]$ filter on the compensation root is answered using *data* extraction. Labels of the compensation root's parent and grandparent are checked using *path*. The rest of the query ($/c$) is answered using the *copy* information. Hence, Q_C is:

$$\pi_{copy/self::*/c}(\sigma_{data>0 \wedge path \sim /b/*}(V))$$

Symbol $\sim$ stands for *match_path* operation that applies regular pattern matching between a linear XPath expression derived from Q^r and *path* extraction. Note that we inverted $parent :: b[parent :: root]$ into $/b/*$. □

If a view does not contain a *reference* extraction, the compensation may not exist, even if the matching algorithm was successful. The compensation construction algorithm detects these cases. If a *reference* extraction is not available, every node in the Q^r has to be marked by the first 3 rules. Otherwise, the compensation cannot be built, which means that the query cannot be answered using the view.

EXAMPLE 4.2 The view $V = //a[b]/c$ with extraction *copy* cannot be used to answer query $Q = //a[b/d]/c/e$. Even though there is an obvious match, the view only contains copies of "c" nodes without their original context. So we cannot check weather sibling node "b" had a "d" child.

Our algorithm will detect this by constructing Q^r = *self::*[parent::*[b/d]]/e.* According to rule 3 *copy* extraction can answer $self :: */e$ part of Q^r, so nodes $self :: *$ and $/e$ of the Q^r are marked as covered by the compensation. Nodes $/b$ and d remain unmarked. Since Q^r contains unmarked nodes, the compensation cannot be constructed, and the view is not usable. □

4.3 Utilizing Multiple Views

Up to now we only considered using a single view and a single compensation root to construct compensation. However, a query can benefit from using multiple views. Similarly the same view could potentially be used multiple times, if the view extraction point mapped to multiple query nodes.

We construct a compensation plan using the following four-step algorithm, which takes as input a set of compensation roots produced by matching one or more views into the query.

1. Find an XPS node in the query tree, that is a lowest common ancestor (LCA) of all compensation roots.

2. For each compensation root q_i , construct an XPath expression Q_i that starts at the compensation root and traverses upward to the LCA node. The Q_i includes local predicates of q_i if the corresponding view contains *data* extraction.

3. Optimize each Q_i, as if it was a compensation, using the algorithm of Section 4.2; construct an intersection of all Q_i expressions.

4. Construct the compensation expression with $\bigcap Q_i$ as the view and LCA as the compensation root.

Note that every view involved in the plan, has to store node references to facilitate upward traversal from the compensation roots to the LCA node.

EXAMPLE 4.3 Consider $Q = //order[@date > "Jan\ 1, 2004"\ and\ lineitem/@price > 100]/number$, and a view $V = //@*$ with $data, path$, and $reference$ extractions. The view maps into the query in two different ways. The two compensation roots are $@date$ and $@price$. Thus the LCA node is $//order$.

In the second step we construct expressions Q_1 and Q_2 which start at the compensation roots and navigate to $//order$. $Q_1 = self :: attribute(date)[. > "Jan\ 1, 2004"]/parent :: order$. $Q_2 = self :: attribute(price)[. > 100]/parent :: lineitem/parent :: order$. Both Q_1 and Q_2 contain local predicates on the compensation roots, because V can answer these predicates directly using the *data* extraction.

The Q_1 and Q_2 are optimized into the expressions P_1 and P_2, shown below, using the three types of extractions stored in the view.

$$P_1 = \pi_{reference/parent::*}(\sigma_{data>"Jan\ 1,2004" \wedge path \sim //order/@date}(V))$$
$$P_2 = \pi_{reference/parent::*/parent::*}(\sigma_{data>100 \wedge path \sim //order/lineitem/@price}(V))$$

Finally, the compensation expression is computed using $//order$ (LCA) as the compensation root and $Q_1 \cap Q_2$ as the view. The resulting plan is:
$\pi_{self::*/number}(P_1 \cap P_2)$ □

Note that the above algorithm provides only one of many ways to construct compensation from multiple materialized views. For some queries and some datasets it might make sense to apply a portion of the compensation before the structural join on the LCA. We are currently investigating cost-based optimization of compensation expressions.

5 Experiments

We implemented the matching framework in the context of XML database research prototype that is under development at IBM Almaden Research Center.

In this section we report on the experimental evaluation of the matching algorithm. We investigated scalability of the algorithm for different classes of queries and view definitions. We do not report compensation construction time, since our algorithm always heuristically picks one compensation to construct, which is relatively cheap. We are currently considering a cost based algorithm, that would construct and estimate costs of various compensations.

Figure 10 shows the performance of the matching algorithm for queries of the following structure: $Q_n = /a[@a_1 = 1\ and\ @a_2 = 2\ and \ldots and\ @a_n = n]$. The query was matched against a view $V = //@*$ resulting in n matches. We varied the value of n from 4 to 64. The bars show the relative[3] time it took to do predicate normalization and matrix construction (matching) steps.

Both predicate normalization and matrix construction time grows linearly with the size of the query. The

[3]The matching and predicate normalization time is reported relative to matching time for the query and view of size one

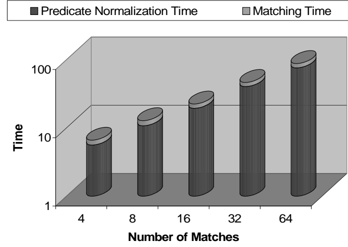

Figure 10: Matching algorithm performance for different sized queries with a single view.

normalization step takes more time since it requires memory allocation to create filter lists and other supporting structures.

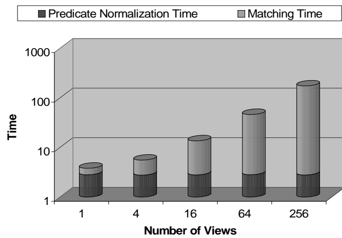

Figure 11: Matching algorithm performance for the same query with a different number of views.

Figure 11 shows that (relative) matching time is a linear function of the number of views defined on the collection of documents that is being queried. For this experiment we used the smallest query expression of the previous experiment: $Q_4 = /a[@a_1 = 1 \textit{ and } @a_2 = 2 \textit{ and } @a_3 = 3 \textit{ and } @a_4 = 4]$. The view definitions where of the form $//@a_k$, where $1 \leq k \leq 4$, so that there was exactly one mapping from each view into the query. Notice that the normalization time plays a much smaller role now, since we normalize the query expression only once.

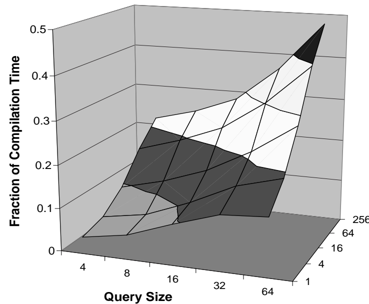

Figure 12: The sum of predicate normalization and matching time as a fraction of total compilation time under various query sizes and number of views.

Figure 12 shows matching time (including predicate normalization), as a fraction of the total query compilation time. The matching time grows linearly in the size of the query and number of views, while the total compiler time grows slower. As a result, the share of the matching algorithm in the total query compilation time increases from 4% in the lower left corner to 48% in the upper right. Note that 64 local predicates may be unrealistically large for a query. It is also unlikely that 256 views are defined over a single document collection. However, some applications do require a large number of views. To address this situation, we are currently exploring pre-filtering techniques that would avoid matching irrelevant views.

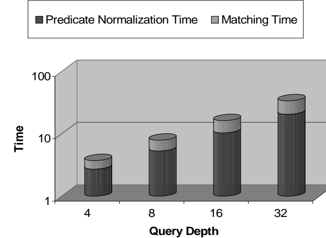

Figure 13: Matching time in case of an exponential number of mappings.

Figure 13 reports the matching algorithm performance for the combinatorial case discussed in Section 3.3, where the match matrix encodes an exponential number of tree mappings. For this experiment the view definition was $V = //a//a//@*$ and the queries were of the form $Q_m = /a[@a_1 = 1]/a[@a_2 = 2]/\ldots/a[@a_m = m]$ resulting in $2 * C_4^m$ possible tree mappings. Our matching algorithm constructs a single match matrix that encodes all of these mappings. The matching time depends almost linearly on the parameter m.

In summary, the overhead of matching for typical queries is not significant. The matching time grows linearly with the number of views defined on a collection. Both predicate normalization and matching time grows almost linearly with the size of the query.

6 Related Work

The problem of rewriting queries using materialized views has been studied extensively in the relational setting [2, 8, 6, 18]. Our XPath matching and compensation algorithms complement this previous work with support for XPath queries.

XPath query containment is a necessary condition for using materialized views, and has recently been studied in [4, 10, 12]. Miklau and Suciu [10] showed that for a subset of XPath containing descendant edges, wildcard tests, and branching, denoted $XP^{\{//,*,[]\}}$, query containment is co-NP complete. Neven and Schwentick [12] showed that adding disjunction to the problem of XPath containment does not increase computational complexity, but did not

provide any algorithms for deciding the containment. They also proved that even with a very simple form of negation, the problem becomes undecidable.

Miklau and Suciu [10] outline an incomplete, but sound and efficient algorithm based on tree mappings, which is able to find containment in vast majority of cases. Our matching algorithm is also based on tree mappings, but considers a richer subset of XPath, including comparison predicates, disjunction, and a full set of axes. They also do not distinguish between next steps and predicates in their XPath representation, as they do not consider disjunctions or comparison predicates.

To the best of our knowledge, the problem of computing compensation to enable rewriting queries using materialized XPath views is not addressed in the literature.

7 Conclusion

We presented a framework for utilizing materialized XPath view in XML Query processing. Our techniques are also applicable in the context of materialized SQL/XML views which contain XML querying functions, such as XMLQuery and XMLExists [15]. The problem of rewriting XML queries using materialized XPath views can be vital for efficient XML query processing, as XML indexes can also be modeled as materialized views.

We addressed two main problems to exploit materialized XPath views: XPath query matching and compensation construction. Our matching algorithm handles a rich subset of XPath, including disjunctions and value-based comparisons. We believe that value based comparison predicates are vital to accelerate processing of XML queries. This is based on the fact that value based comparison predicates are in general more selective than existential structural tests, and a B^+-tree index on typed data values might be exploited to further speed up evaluation of such predicates. The matching algorithm records all mappings which are later used to construct compensation.

The algorithm has polynomial time complexity in the size of the view and query. Moreover, the experimental study shows that in most cases matching time is actually linear in the size of the input.

We also provided algorithms to compute compensation. We investigated exploiting different kinds of extractions in compensation expressions and described heuristics to exploit these extraction types.

In the future, we plan to investigate a cost-based compensation construction, that will produce a number of compensation plans and chose the one based on a cost estimate. Using this cost model, we also plan to investigate algorithms to choose the most effective set of materialized XPath views given a query workload.

We also plan to employ this framework in a larger scope of rewriting XQuery using materialized XQuery views.

References

[1] ISO/IEC 9075-14:2003. Information technology – database languages – sql – part 14: Xml-related specifications (sql/xml).

[2] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of ICDE*, pages 190–200, 1995.

[3] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings of VLDB*, pages 341–350, Roma, Italy, 2001.

[4] A. Deutsch and V. Tannen. Containment and integrity constraints for xpath. In *Proceedings of KRDB*, 2001.

[5] R. Goldman and J. Widom. Dataguides:enabling query formulation and optimization in semistructured databases. In *Proceedings of VLDB*, pages 436–445, 1997.

[6] J. Goldstein and P. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of SIGMOD*, Santa Barbara, CA, 2001.

[7] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of SIGMOD*, 2002.

[8] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of PODS*, pages 95–104, 1995.

[9] Quanzhong Li and Bongki Moon. Indexing and querying xml data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB)*, pages 361–370, Roma, Italy, September 2001.

[10] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *Proceedings of PODS*, pages 65–76, 2002.

[11] S. Nestorov, J. D. Ullman, J. L. Wiener, and S. S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *Proceedings of ICDE*, pages 79–90, 1997.

[12] F. Neven and T. Schwentick. Xpath containment in the presence of disjunction, dtds and variables. In *Proceedings of ICDT*, 2003.

[13] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward. In *Workshop on XML-Based Data Management*, 2002.

[14] F. Rizzolo and A. O. Mendelzon. Indexing xml data with toxi. In *Proceedings of WebDB*, pages 49–54, 2001.

[15] SQL/XML. See http://www.sqlx.org.

[16] *XML Path Language (XPath) Version 2.0*, November 2003. W3C Working Draft, See http://www.w3.org/TR/xpath20.

[17] *XQuery 1.0: An XML Query Language*, November 2003. W3C Working Draft, See http://www.w3.org/TR/xquery.

[18] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *Proceedings of SIGMOD*, pages 105–116, 2000.