Querying transformed XML documents: Determining a sufficient fragment of the original document

Sven Groppe , Stefan Böttcher

University of Paderborn
Faculty 5 (Computer Science, Electrical Engineering & Mathematics)
Fürstenallee 11 , D-33102 Paderborn , Germany
email : sg@uni-paderborn.de , stb@uni-paderborn.de

Abstract. Large XML documents which are stored in an XML database can be transformed further by an XSL processor using an XSLT stylesheet. In order to answer an XPath query based on the transformed XML document, it may be of considerable advantage to retrieve and process only that part of an XML document stored in the database which is used by a query. Our contribution uses an XSLT stylesheet to transform a given XPath query such that the amount of data which is retrieved from the XML database and transformed by the XSL processor according to the XSLT stylesheet is reduced.

1 Introduction

1.1 Problem origin and motivation

Whenever XML data is shared by heterogeneous applications, which use different XML representations of the same XML data, it is necessary to transform XML data from one XML format 1 into another XML format 2. The conventional approach is to transform entire XML documents into the application-specific XML format, so that each application can work locally on its preferred format. Among other things, this causes problems of replication (especially synchronization problems), consumes a lot of processing time and in distributed scenarios, leads to high transportation costs. A more economic approach to the integration of heterogeneous XML data involves transforming and transporting the data on demand only, and only the amount which is needed to perform a given operation.

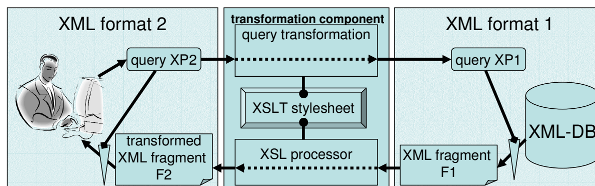

Figure 1: The transformation process

More specifically, our work is motivated by the development of an XML database system which ships data to remote clients. Whenever clients use their own XML format, XSLT is used for transforming documents given in XML format 1 (of the database) into XML format 2 (of the client). Whenever a client application submits an XPath query XP2 for XML data in format 2, we propose transforming XP2 using a new query transformation algorithm into an XPath query XP1 on the original XML data in format 1. The evaluation of XP1 yields a fragment of the original document which, when transformed using the XSLT stylesheet, can be used to evaluate the original query XP2 of the client. This approach (cf. Figure 1) may result in a considerable reduction in the amount of data transformed and shipped in comparison to the process of transforming the whole document via the XSLT stylesheet and applying the query XP2 afterwards.

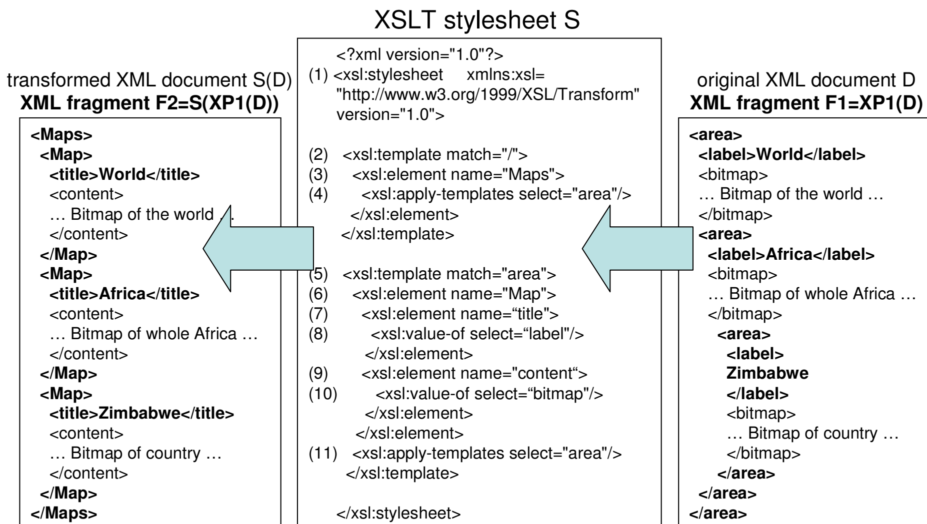

Figure 2: Example of the transformation of F1 into F2 by an XSLT stylesheet S

For example, consider the XML document D and the XSLT stylesheet S in Figure 2. The XML document D contains named maps (in the form of bitmaps with large sizes) of nested areas. The XSLT stylesheet S transforms the XML document D to S(D), a flat presentation of the XML document. We only retrieve the titles of the maps by applying an XPath query

```
XP2 = /Maps/Map/title
```

given in XML format 2 on S(D). Throughout this paper, we explain why it is sufficient to only transform that bold face part of XML document D (i.e. XML fragment F1) in Figure 2, which can be described using the following query XP1 given in XML format 1

```
XP1 = /area (/area)* /label
```

where A* is a short notation for an arbitrary number of paths A.[1] The result of only transforming the XML fragment F1 is the XML fragment F2, which is the bold face part of S(D) in Figure 2. Notice, that F2 is still sufficient to answer the query XP2, but it excludes especially the bitmaps with large sizes.

The algorithmic problem is as follows: Given an XPath query XP2 and an XSLT stylesheet S, which are used to transform XML documents D (e.g. the XML document in Figure 2), or XML fragments respectively, into S(D), we compute an XPath query XP1 such that the following property holds:

We retrieve the same result for all XML documents D given in format 1,

- when firstly we apply the XSLT stylesheet S to D, and then apply the query XP2 to the XML fragment S(D), and
- when firstly we apply the query XP1 to the XML fragment D, then transform the result according to the XSLT stylesheet S and finally apply the query XP2,

i.e. XP2(S(D)) must be equivalent to XP2(S(XP1(D))). Our goal is to keep F1=XP1(D) small in comparison to D.

In this case, we can ship and transform F1=XP1(D) instead of D, which saves transportation costs and processing time.

1.2 Relation to other work and our focus

For the transformation of XML queries into queries to other data storage formats at least two major research directions can be distinguished: firstly, the mapping of XML queries to object oriented or relational databases (e.g. [BBB00]), and secondly, the transformation of XML queries or XML documents into other XML queries or XML documents (e.g. [Ab99]). We follow the second approach; however, we focus on XSL [W3C01] for the transformation of both, data and XPath [W3C99] queries.

Within related contributions to schema integration, two approaches to data and query translation can be distinguished. While the majority of contributions (e.g. [CDSS98], [ACM97], [SSR94]) map the data to a unique representation, we follow [CG00] and [CG99] and map the queries to those domains where the data resides.

[CVV01] reformulates queries according to path-to-path mappings. We go beyond this, as we use XSLT as a more powerful mapping language.

---

[1] Standard XPath evaluators do not support A*, but we can retrieve a superset by replacing A*/ with //. Furthermore, a modified XPath evaluator has to return not only the result set of XP1 (as standard XPath evaluators do), but a result XML fragment F1. This result XML fragment F1 must contain all nodes and all their ancestors up to the root of the original XML document D, which contribute to the successful evaluation of the query XP1.

[Mo02] describes how XSL processing can be incorporated into database engines, but it focuses on efficient XSL processing.

In contrast to all the other approaches, we focus on the transformation of XPath queries according to a mapping, which is implicitly given by an XSLT stylesheet.

1.3 Considered subsets of XPath and XSLT

Since XPath and XSLT are very powerful and expressive languages, however, our applications only need a small subset. We currently restrict XPath queries `XP2`, such that they conform to the following rule for `LocationPath` given in the Extended Backus Naur Form (EBNF):

```
LocationPath      ::= (("/" | "//") Name)*.
```

This subset of XPath allows for the querying for an XML fragment which can be described by succeeding elements (in an arbitrary depth).

Similarly, we restrict XSLT, i.e., we consider the following nodes of an XSLT stylesheet:

- `<xsl:stylesheet>`,
- `<xsl:template match=M1 name=N>`,
- `<xsl:element name=N>`,
- `<xsl:apply-templates select=S1>`,
- `<xsl:text>`,
- `<xsl:value-of select=S2>`,
- `<xsl:for-each select=S1>`,
- `<xsl:call-template name=N>`,
- `<xsl:if test=T>`,
- `<xsl:choose>`,
- `<xsl:when test=T>`,
- `<xsl:otherwise>`,
- `<xsl:processing-instruction>`,
- `<xsl:comment>` and
- `<xsl:sort>`,

where `S1`, `S2` and `M1` contain an XPath expression with relative paths without function calls, `T` is a boolean expression with relative paths and `N` is a string constant. Additionally, `M1` can contain the document root "`/`".

Whenever attribute values are generated by the XSLT stylesheet, we assume (in order to keep this presentation simple) that this is only done in one XSLT node (i.e. `<xsl:text>` or `<xsl:value-of select=S2>`).

2 Query transformation as search problem in the stylesheet graph

The Querying of the transformed XML document S(D) using a given query XP2 only selects a certain part of S(D) (i.e. XP2(S(D))), which is generated by the XSLT processor at certain so called *output nodes* of the XSLT stylesheet S. In the example of Figure 2, all the elements Maps in S(D) are generated by the node (3) of S (see Figure 2), all elements Map are generated by node (6) and all elements title and their contents are generated by node (7) and (8). These output nodes of the XSLT stylesheet S are reached, after a sequence of nodes (which we call *stylesheet paths*) of the XSLT stylesheet S have been executed. In the example, one stylesheet path that contains the nodes (3), (6), (7) and (8) is <(1),(2),(3),(4),(5),(6),(7),(8)>. While executing these stylesheet paths, the XSLT processor also processes so called *input nodes* (e.g. node (4) and (8)) each of which selects a node set of the input XML document D. The input nodes altogether select a certain whole node set of the input XML document D. In the stylesheet path above, this is the node set /area/label. When considering our idea to reduce the amount of data of the input XML document, we notice that all the nodes (but *not* more nodes!) of the input XML document which are selected within input nodes along the stylesheet path must be available in order to execute the stylesheet path in the same way as all nodes of the input XML document are available. If we can determine the whole node set (described using a query XP1), which is selected on *all* stylesheet paths, which generate output which fits to the query XP2, we can then select a smaller, but yet sufficient part XP1(D) of the input XML document D, where the transformed XP1(D), i.e. S(XP1(D)), contains all the information required to answer the query XP2 correctly, i.e. XP2(S(XP1(D))) is equivalent to XP2(S(D)).

Within our approach, at first we transform the XSLT stylesheet into a stylesheet graph (see Section 2.1 and 2.2) in order to search more easily for stylesheet paths (see Section 2.3), which generate elements and their contents in the correct order according to the query XP2.

For each of these stylesheet paths, within Section 3 we determine the so called *input path expression* of the XSLT stylesheet, which summarizes the XPath expressions of the input nodes along the stylesheet path. The transformed query XP1 is the disjunction of all the determined input path expressions of each stylesheet path.

2.1 Determination of the callable templates

For the construction of the stylesheet graph (see section 2.2.), we have to determine (a superset of) all the templates <xsl:template match=m> which can (possibly) be called from a node <xsl:apply-templates select=s>.

Within the node `<xsl:apply-templates select=s>` a certain node set is selected depending on its context, where `s` contains a relative path (see section 1.3). We ignore the exact context of the node here and describe a superset `s_super` of the selected node set by assigning `//s` to `s_super`. Similarly, if `m`<>"`/`" we assign //`m` to `m_super` for the node `<xsl:template match=m>`, which describes a superset of the matching nodes `m`. If `m`="`/`" we assign the document root "`/`" to `m_super`. For example, see nodes (4) and (5) of Figure 2. Within this example, `s_super` is `//area`, `m_super` is `//area`.

We can then use a fast (but incomplete) tester (e.g. the one in [BT03]) in order to prove that `m_super` and `s_super` are disjointed. Whenever the supersets `s_super` and `m_super` are disjointed, we are then sure that `s` and `m` are also disjointed, i.e. `<xsl:apply-templates select=s>` can not call a template `<xsl:template match=m>`. For example, this is the case for node (4) and node (2) of Figure 2. If the intersection of `s_super` and `m_super` is not empty, we must consider the fact that the template can possibly match the selected node set. For example, this is the case for `s_super=//area` of node (4) and `m_super=//area` of node (5) of Figure 2.

Since this can give us a superset of the templates which can be applied, the transformed query `XP1` may query for more than is needed. Note however that we never obtain a wrong result, because we always apply the query `XP2` afterwards.

2.2 Stylesheet graph

In order to compute the node set of the input XML document which is relevant to the query `XP2`, we transform an XSLT stylesheet (e.g., that of Figure 2) into a graph (e.g., that of Figure 3). The basic idea involves connecting *all* nodes `n1` and `n2` by an edge, if `n2` can be reached directly after `n1`, while executing the XSLT stylesheet.

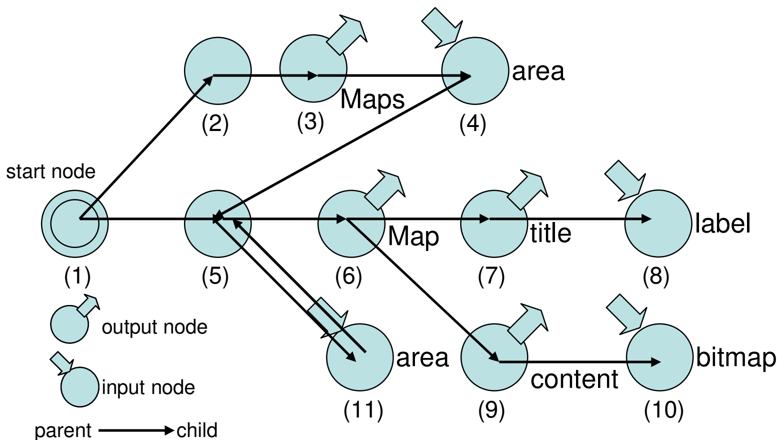

Figure 3: Stylesheet graph of the XSLT stylesheet S of figure 2

A *stylesheet graph* consists of a set N of *nodes* and a set E of directed *edges*. A node n ∈ N is a *normal node*, an *output node* or an *input node*. An output node contains an additional entry A which represents the XML element (e.g. Map) that is generated by the node during the transformation process of the XML document. An input node contains an additional XPath expression entry which represents the read operations on the input XML document during the transformation. One special node of the stylesheet graph is the start node. An edge e is a pair of nodes, e=(n1,n2) with n1,n2 ∈ N.

The following rules transform an XSLT stylesheet into the corresponding stylesheet graph:

a. For each node in the XSLT stylesheet, we insert an own node into the stylesheet graph. In the example, the numbers below the nodes of the stylesheet graph of Figure 3 correspond to the numbers of the nodes in the XSLT stylesheet of Figure 2. For example the node (1) in Figure 3 corresponds to the node <xsl:stylesheet ...> in the XSLT stylesheet of Figure 2.
b. The node in the stylesheet graph that corresponds to the node <xsl:stylesheet> of the XSLT stylesheet is the *start node* of the stylesheet graph. For example, see node (1) in Figure 2 and Figure 3.
c. For each node in the stylesheet graph we check, whether or not the node belongs to the *output nodes* or to the *input nodes*:
   1) If the corresponding node in the XSLT stylesheet generates an element E (<xsl:element name=E>), the node in the stylesheet graph belongs to the *output nodes*: We assign E which is generated in the corresponding node of the XSLT stylesheet to the output entry of the output node. For example, see nodes (3), (6), (7) and (9) in Figures 2 and 3.
   2) If the corresponding node in the XSLT stylesheet selects a node set S of the input XML document (<xsl:apply-templates select=S/>, <xsl:value-of select=S/>, or <xsl:for-each select=S>), the node in the stylesheet graph belongs to the *input nodes*: we copy S to the input entry of the node of the stylesheet graph. For example, see nodes (4), (8) and (10) in Figure 2 and Figure 3. The same applies to <xsl:if test=T> or <xsl:when test=T>, if S occurs in the Boolean expression T.
d. Let n1 and n2 be the nodes in the stylesheet graph which correspond to the nodes S1 and S2 in the XSLT stylesheet. We draw an edge from n1 to n2, if
   1) S2 is a child node of S1 within the XSLT stylesheet (for example, see node (1) and (2) in Figure 2 and Figure 3), or
   2) S1 is a node <xsl:call-template name=N> and S2 a node <xsl:template name=N> with an attribute name set to the same N, or
   3) S1 is a node <xsl:apply-templates select=s/> and S2 a node <xsl:template match=m> and the template of S2 can possibly be called from the selected node set s (see section 2.1). For example, see nodes (4) and (5) in Figure 2 and Figure 3.

2.3 Output path search in the stylesheet graph

Algorithm 1 contains the (depth-first search) algorithm of the output path search. We describe the idea behind the algorithm in this section:

In order to determine the paths through an XSLT stylesheet graph which may generate output that is relevant to `XP2`, we search for so called *successful element stylesheet paths*, i.e. paths which begin at the start node and contain all the output nodes of the stylesheet graph which may contribute to answering the query `XP2`.

For example, for `XP2=/Maps/Map/title` and the XSLT stylesheet of Figure 2 (or its stylesheet graph shown in Figure 3, respectively), we search for the output nodes (see Algorithm 1, lines 36 to 38) which generate the elements `Maps`, `Map` and `title` in the correct order. Firstly, we begin our search at the start node (1) and we search for an output node which generates `Maps`. The search can pass normal nodes and input nodes as they do not generate any output, which does not fit to `XP2` (see Algorithm 1, lines 33 to 35). The search can also pass any output nodes if we search next for an element `E` in arbitrary depth, i.e. for `//E` (see Algorithm 1, lines 33 to 35). We find this output node generating the element `Maps` at the node (3) after the nodes (1) and (2). Afterwards, we search for an output node which generates `Map`. We find an output node (6) generating `Map`, after the nodes (4) and (5) have been passed. The following node (7) generates `title` (and node (8) its content), i.e. the last element in `XP2` to be searched for: We found a successful element stylesheet path with nodes (1), (2), (3), (4), (5), (6), (7) and (8).

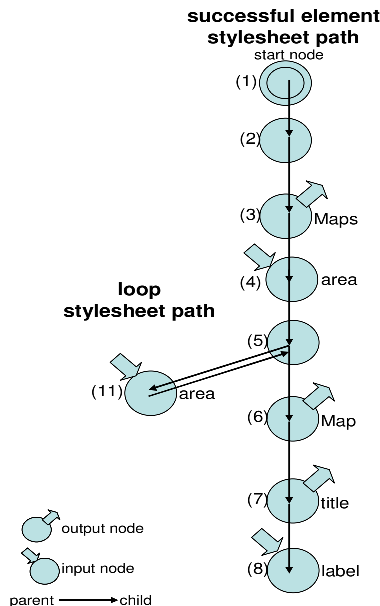

Figure 4: Result of the Output Path Search

```
(1)   types:
(2)     list of (Node, XPath)                      Stylesheet_path;
(3)   global variables:
(4)     Stylesheet_graph                                        sg;
(5)     list of Stylesheet_path
(6)                         successful_element_stylesheet_paths;
(7)     list of ((Node, XPath), Stylesheet_path)
(8)                                       loop_stylesheet_paths;
(9)
(10)  startSearch(in XPath XP2)
(11)  { SearchElement(sg.getStartNode(),XP2,
(12)                 new Stylesheet_path()); }
(13)
(14)  boolean isLoop(in Node N, in XPath XP2r,
(15)                inout Stylesheet_path sp)
(16)  { if (sp.contains( (N,XP2r) ))
(17)    { loop_stylesheet_paths.add( (N,XP2r),
(18)         sp.subList(sp.firstOccurrence((N,XP2r))+1,
(19)                                         sp.size()) );
(20)      return true;
(21)    }
(22)    else { sp.add( (N,XP2r) );
(23)           return false;
(24)         }
(25)  }
(26)
(27)  SearchElement(in Node N, in XPath XP2r,
(28)               in Stylesheet_path sp)
(29)  { if(not isLoop(N, XP2r, sp) )
(30)    { if(XP2r is empty and
(31)         (N is output node or N has no descendant))
(32)           successful_element_stylesheet_paths.add(sp);
(33)      if(N is not output node or XP2r starts with "//")
(34)        for all descendants DN of N do
(35)           SearchElement(DN, XP2r, sp);
(36)      if( N is output node generating element E and
(37)         ( XP2r starts with "/E" or "//E" ))
(38)      { XP2r=XP2r.stringAfter("E");
(39)        if(XP2r is empty and N has no descendant)
(40)          successful_element_stylesheet_paths.add(sp);
(41)        for all descendants DN of N do
(42)           SearchElement(DN,XP2r,sp);
(43)      }
(44)    }
(45)  }
```

Algorithm 1: Output path search

In order to store information for each part of the query XP2 which we search next, we define a s*tylesheet path* as a list of pairs (N, XP2r) where N is a node in the stylesheet graph and XP2r is the remaining location steps of XP2 which still have to be processed (see Algorithm 1, line 2). We call the stylesheet path, which contains all the visited nodes of the path from the start node to the current node in the visited order, the *current stylesheet path* sp.

During the search it may occur, that we revisit a node N of the XSLT graph without any progress in the processing of XP2r. For example, we can visit the nodes (1), (2), (3), (4), (5), (11) and then the node (5) again in Figure 3. We call this a *loop*, and we define a loop as follows: The loop is the current stylesheet path minus the stylesheet path of the first visit of N. In the example, this is <((11), /Map/title), ((5), /Map/title)> in Figure 4. For each loop in the stylesheet graph (see Algorithm 1, lines 14 to 25), we store the loop itself, the current node N and XP2r as an entry to the set of *loop stylesheet paths*, because we need to know the input which is consumed when the XSLT processor executes the nodes of a loop (see Section 3.4). In order to avoid an infinite search, we abort the search at this point.

Figure 4 shows both, the successful element stylesheet path and the attached loop stylesheet path of our example.

3 Computing input path expressions

Within Section 2 we computed successful element stylesheet paths such that (only) when the XSLT processor tracks a successful element stylesheet path (and its attached loop stylesheet paths), does it generate an XML fragment F2 which contributes to the query XP2. While tracking a successful element stylesheet path, the XSLT processor selects a certain node set called *input node set* of the input XML document whose existence is necessary for the execution of the successful element stylesheet path. The input node set is described using the so called *input path expressions*, which are contained in the input entries of the input nodes. The remaining task to be completed is to determine this input node set and to describe this input node set using a query XP1.

The XSLT processor does not select the input node set of the input XML document immediately. In fact, the XSLT processor selects the input node set step by step in different input nodes of the XSLT stylesheet which are described by their input path expressions in the successful element stylesheet path and its attached loop stylesheet paths. For this reason, we have to combine all these input path expressions along a successful element stylesheet path (and its attached loop stylesheet paths). Figure 5 shows the computation of the input path expressions of our example, which we will explain in more detail in the following subsections.

For example (see Figure 5), the input path expression `/` (selecting the document root) is matched within node (2) and the document root is also the current input node set of node (3). The current input node (4) selects a relative input path expression `area`, so that the total selected input path expression is `/area` after the current input node (4). We use a variable *current input path expression (current ipe)* in order to collect the currently selected input path expression. The *current ipe* contains a combination of all the input path expressions of all input nodes up to (and including) the current node.

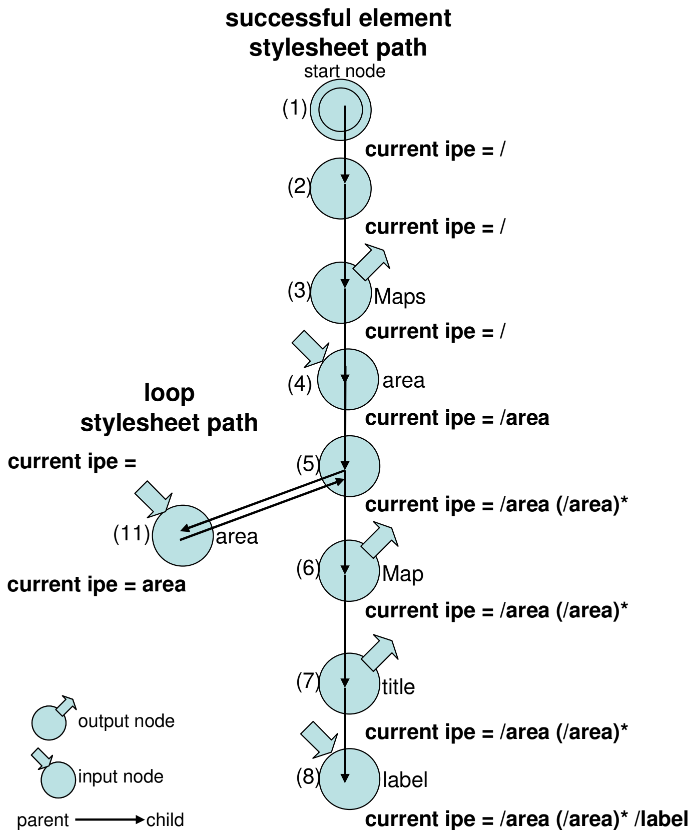

Figure 5: Computing the input path expression of the running example

We mainly iterate through each successful element stylesheet path and we

- compute the new current ipe ($\texttt{current ipe}_{\texttt{new}}$) from the input path expression of the current node and the old current ipe ($\texttt{current ipe}_{\texttt{old}}$).
- recursively compute and combine current ipes of attached loop stylesheet paths.

The initialization of current ipe is described in Section 3.1. The different combination steps are described in Sections 3.2 to 3.4, and the determination of the complete input path expression is described in Section 3.5.

3.1 Initialization of current ipe

In general, the current ipe in each successful element stylesheet path is initialized using the match attribute of that node within the XSLT stylesheet, that corresponds to the second node of this successful element stylesheet path (the first node always corresponds to a node `<xsl:stylesheet>`, the second to a node `<xsl:template match=m>`). However, if `m` (and therefore the current ipe) contains a relative path (i.e. `m` does not contain the document root `/`), we replace `m` with `//m` within the current ipe in order to complete the initialization. As an XML node with an arbitrary depth can be matched with a template because of built-in templates, we do this when the value of the match attribute contains a relative path.

In our example (see Figure 5), current ipe is initialized with the document root `/` before node (2).

3.2 Non-input nodes

Whenever a node is neither an input node nor a node with an attached loop stylesheet path, then the current ipe remains unchanged, i.e., it is identical to its previous value.

In our example (see Figure 5), this is the case for the nodes (2), (3), (6) and (7).

3.3 Basic combination step

Figures 5 shows three examples (see nodes (4), (11) and (8)) of the computation of a new current input path expression ($\texttt{current ipe}_{new}$) of input nodes from an old current input path expression ($\texttt{current ipe}_{old}$).

The general rule is as follows:

Let `r` be the input path expression of the current input node. The current ipe must be combined with `r`:

$$\texttt{current ipe}_{new} = \texttt{current ipe}_{old} \; / \; \texttt{r}$$

3.4 Loop combination step

In our example of Figure 5, the loop stylesheet path `<((11),/Map/title),((5),/Map/title)>` is attached to the node (5). Within the loop stylesheet path, the node set `area` is selected. While tracking the successful element stylesheet path, the XSLT processor can execute the nodes of the loop stylesheet path an arbitrary number of times. This induces the XSLT processor to select the node set `area`, i.e. `(/area)*` an arbitrary number of times. As the current ipe before the node (5) is `/area`, the current ipe after the node (5) is `/area (/area)*`.

The general rule is as follows:

If there is a loop stylesheet path attached to the current node (for example, see node (5) with the loop stylesheet path <((11),/Map/title),((5),/Map/title)> in Figure 5), we start an additional recursive computation of the input paths of this loop stylesheet path. Before this recursive computation begins, we initialize the current input path expression ($\mathtt{current\ ipe}_{\mathtt{loop}}$) of the loop with an empty path. Then we recursively[2] compute in the loop as before and obtain the current ipe after the last node of the loop ($\mathtt{current\ ipe}_{\mathtt{end\ of\ loop}}$). We compute $\mathtt{current\ ipe}_{\mathtt{new}}$ of the node, to which the loop is attached, according to the following rules:

In every iteration of the loop, $\mathtt{current\ ipe}_{\mathtt{end\ of\ loop}}$ is selected in the context of the input path expression $\mathtt{current\ ipe}_{\mathtt{old}}$:

$$\mathtt{current\ ipe}_{\mathtt{new}} = \mathtt{current\ ipe}_{\mathtt{old}}\ (\mathtt{/current\ ipe}_{\mathtt{end\ of\ loop}})^*$$

Let us assume that there are n>1 loops attached to the current node. Then we compute the current ipe after the last node of the loop ($\mathtt{current\ ipe}_{\mathtt{end\ of\ loop}}\mathtt{[i]}$) for each loop i. Then we compute $\mathtt{current\ ipe}_{\mathtt{new}}$ for multiple loops using the following equation:

$$\mathtt{current\ ipe}_{\mathtt{new}} = \mathtt{current\ ipe}_{\mathtt{old}}\ (\mathtt{/current\ ipe}_{\mathtt{end\ of\ loop}}\mathtt{[1]}\ |\ \ldots\ |\ \mathtt{/current\ ipe}_{\mathtt{end\ of\ loop}}\mathtt{[n]})^*$$

3.5 The complete input path expression **XP1**

The complete input path expression which is used as query XP1 on the input XML document is the union of all the current ipes after the last node of each of the n successful element stylesheet paths (1..n),

$$\mathtt{XP1} = \mathtt{current\ ipe}_1\ |\ \ldots\ |\ \mathtt{current\ ipe}_n.$$

where $\mathtt{current\ ipe}_{\mathtt{x}}$ is the current ipe after the last node of the x-th successful element stylesheet path has been processed.

If there is no entry in the successful element stylesheet path (i.e. n=0), then XP1 remains empty.

Within our example of Figure 5, there is only one entry in the set of successful element stylesheet paths, and XP1 is equal to the current ipe after the last node (8):

```
XP1   =     /area (/area)* /label
```

---

[2] Note that a loop can contain other loops.

3.6 Result of the XPath evaluator for XP1

The XPath evaluator which evaluates the XPath expression XP1 on the XML database produces an optimal result, if it supports the newly introduced A* operator which is a short notation for an arbitrary number of location steps A. If the XPath evaluator does not support the A* operator, then the XPath evaluator can return a superset by simply replacing A*/ with //.

In order to determine the resulting XML fragment of the query XP1, a modified XPath evaluator has to return not only the result set of XP1 (as standard XPath evaluators do), but a result XML fragment F1. This result XML fragment F1 must contain all nodes and all their ancestors up to the root of the original XML document D, which contribute to the successful evaluation of the query XP1.

For example, the evaluation of the XPath expression

```
XP1 = /area (/area)* /label
```

on the XML database will result in the XML fragment F1 of Figure 2, which is the bold face part of the XML document D.

4 Summary and Conclusions

In order to reduce data transformation and data transportation costs, we compute a transformed query XP1 from a given query XP2 and a given XSLT stylesheet which can be applied to the original XML document. This allows us to retrieve a smaller, but yet the sufficient fragment F1 which contains all relevant data. F1 can be transformed by the XSLT stylesheet into F2, from which the query XP2 selects the relevant data.

In comparison to other contributions to query reformulation, we transform the XSLT stylesheet into a stylesheet graph, which we use in order to search for paths according to the given query XP2. This allows us to transform the given query XP2 into a query XP1 on the basis of input path expressions which are found in input nodes along the searched path.

We expect our approach to queries on transformed XML data to have considerable advantages over the standard approach which transforms the entire XML document particularly for very large XML documents and for shipping XML data to remote clients.

Our approach enables the seamless incorporating of XSL processing into database management systems, which in our opinion will become increasingly important in the very near future.

An extension of the approach presented here which would involve supporting a larger subset of XPath and XSLT would appear to be very promising.

Acknowledgements

This work is funded by the MEMPHIS project (IST-2000-25045).

References:

[Ab99] S. Abiteboul, On views and XML. In *PODS*, pages 1-9, 1999.

[ACM97] S. Abiteboul, S. Cluet, and T. Milo, Correspondence and translation for heterogeneous data. In *Proc. of the 6th ICDT,* 1997.

[BG03] S. Böttcher, and S. Groppe, Automated Data Mapping for Cross Enterprise Data Integration. *International Conference of Enterprise Information Systems (ICEIS 2003)*, Angers, France, 2003.

[BT03] S. Böttcher, and A. Türling, Checking XPath Expressions for Synchronization, Access Control and Reuse of Query Results on Mobile Clients. *Workshop: Database Mechanisms for Mobile Applications*, Karlsruhe, Germany, 2003.

[BBB00] R. Bourret, C. Bornhövd, and A.P. Buchmann, A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases. *2nd Int. Workshop on Advanced Issues of EC and Web-based Information Systems (WECWIS)*, San Jose, California, 2000.

[CG99] C.-C. K. Chang, and H. Garcia-Molina, Mind your vocabulary: Query mapping across heterogeneous information sources. In *Proc. of the 1999 ACM SIGMOD Conf.*, Philadelphia, 1999. ACM Press, NY.

[CG00] C.-C. K. Chang, and H. Garcia-Molina, Approximate Query Translation Across Heterogeneous Information Sources. *VLDB 2000, 2000.*

[CDSS98] S. Cluet, C. Delobel, J. Simon, and K. Smaga, Your mediators need data conversion! In *Proc. of the 1998 ACM SIGMOD Conf., 1998.*

[CVV01] S. Cluet, P. Veltri, and D. Vodislav, Views in a Large Scale XML Repository. In *Proceedings of the 27th VLDB Conference*, Roma, Italy, 2001.

[LW00] A.Y. Levy, and D.S. Weld, Intelligent internet-systems. *Artificial Intelligence*, 118(1-2), 2000.

[Mo02] G. Moerkotte, Incorporating XSL Processing Into Database Engines. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002.

[SSR94] E. Sciore, M. Siegel, and A. Rosenthal, Using semantic values to facilitate interoperability among heterogeneous information systems. *Trans. on Database Systems*, 19(2), 1994.

[W3C01] W3C, Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL/, 2001.

[W3C99] W3C, XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath/, 1999.