Incorporating XSL Processing Into Database Engines

Guido Moerkotte

University of Mannheim
D7, 27
68131 Mannheim
Germany
moerkotte@informatik.uni-mannheim.de

Abstract

The two observations that 1) many XML documents are stored in a database or generated from data stored in a database and 2) processing these documents with XSL stylesheet processors is an important, often recurring task justify a closer look at the current situation. Typically, the XML document is retrieved or constructed from the database, exported, parsed, and then processed by a special XSL processor. This cumbersome process clearly sets the goal to incorporate XSL stylesheet processing into the database engine.

We describe one way to reach this goal by translating XSL stylesheets into algebraic expressions. Further, we present algorithms to optimize the template rule selection process and the algebraic expression resulting from the translation. Along the way, we present several undecidability results hinting at the complexity of the problem on hand.

1 Introduction

This work is motivated by three observations. The first observation is that with the rapid proliferation of XML [6], more and more XML documents are

- stored in databases or
- generated from data stored in a database.

Today, all major vendors of relational database systems provide facilities to manage and generate XML documents. Among the most important relational systems are IBM's DB2 [1], Microsoft's SQL Server [2], and Oracle [3] by Oracle Corp.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

The second observation is that XML documents are rarely retrieved for their own sake but more often subsequently processed by some application. The most prominent application is an XSL [10] stylesheet processor. In one usage scenario, a web server runs an XSL processor to generate HTML or XHTML documents out of XML documents, which are then served to the web.

The third observation concerns the coupling of the database management system and the XSL stylesheet processor. We found the following to be representative. First, an XML document is retrieved from a database or constructed from data stored in a database. The result is an XML document in textual form, which is then exported from the DBMS and written to a file. This file is then parsed and processed by the XSL stylesheet processor.

This cumbersome process raises the question whether it is possible to incorporate stylesheet processing into the database engine. If we can do so, we gain several advantages. The most obvious one is that the intermediate materialization becomes superfluous. Further, if the XSL stylesheet accesses only parts of a document, only these parts have to be retrieved/constructed. Note that in the above approach, the whole document has to be retrieved/constructed and passed to the XSL processor since the DBMS has no knowledge about the stylesheet. Last, let us consider the case where the XML document is constructed from the database. In this case, an XML document is specified declaratively by some construction specification—for example in an extended SQL [1, 2, 3, 17, 19]. One can imagine that a combined optimization of the construction specification and the XSL stylesheet leads to significant performance improvements.

The goal of the paper is to show one way to incorporate XSL stylesheet processing into database engines. We have chosen to translate XSL stylesheets into algebraic expressions. The reason is that XML construction is also based on translating some extended SQL into algebraic expressions which are then optimized (e.g. [19]).

Our contributions can be summarized as follows. We show how to extend an algebra to be able to capture XSL stylesheet processing. At the same time, we take care that these extensions can be integrated easily into current

database engines where physical algebraic operators are implemented using the iterator principle [13]. We provide an algorithm to translate the core of XSL into our algebra and optimize the resulting algebraic expression. A large fraction of the paper is devoted to the problem of optimizing the process of selecting the next applicable template rule. Along our way, we present several new undecidability results hinting at the complexity of the discussed problem.

The rest of the paper is organized as follows. The next section summarizes XSL essentials. The material on template rule selection and its optimization is contained in Section 3. Section 4 introduces the algebra and gives an algorithm to translate an XSL stylesheet into the algebra. Section 5 shows how the initial algebraic expression can be optimized. In particular, it shows how recursion can be eliminated. Section 6 concludes the paper.

2 XSL Essentials

General XSL [10] is a language for representing stylesheets. According to the W3C, it consists of three parts: XSL Transformation (XSLT [7]) for describing the transformation of XML documents, XML Path Language (XPath [8]) for accessing parts of an XML document, and XSL Formatting Objects for specifying formatting semantics (a description is contained in the document on XSL). Since we are not interested in formatting and we can easily incorporate XSL formatting elements into our approach, we concentrate on XSLT, which contains the logic of any XSL stylesheet. For space reasons, we cannot give a summary of XPath or describe XSLT in full detail. The reader is referred to the specification documents.

XSLT is a language for transforming XML documents into other documents. Such a transformation is expressed in XSLT by means of a stylesheet consisting of - among other things - a sequence of *template rules*. These consist of two parts: a *pattern* which is matched against nodes in the source document and a *template* which can be instantiated to form part of the result document. A template is instantiated for a particular source node that matches the pattern of the template rule. When a pattern is instantiated, each instruction is executed and replaced by the result it creates. The results of all instructions are concatenated to form the result of the instantiated template. Instructions can select and process descendant source nodes.

Syntax The syntax of XSLT is XML. Figures 1-4 contain a sample DTD and XML document, a stylesheet, and the result of applying the stylesheet to the document. We use the `xsl` namespace to distinguish XSLT elements from other elements. A stylesheet is always embedded in an `xsl:stylesheet` element. A template rule is expressed by an `xsl:template` element. Its `match` attribute contains the pattern in the form of an XPath expression. A template can also contain a `priority` attribute whose value must be a number. If no priority is specified, a default priority is assigned (see Fig. 5).

Inside the `xslt:template` element we find literal XML or text as well as XSLT instructions. Among the

```
<!ELEMENT world (country*)>
<!ATTLIST world id ID #REQUIRED>
<!ELEMENT country (city*)>
<!ATTLIST country
  id   ID #REQUIRED
  name CDATA #REQUIRED>
<!ELEMENT city EMPTY>
<!ATTLIST city
  id   ID #REQUIRED
  name CDATA #REQUIRED>
```

Figure 1: A Sample DTD

```
<?xml version="1.0"?>
<!DOCTYPE world SYSTEM "world.dtd">
<world id="1">
  <country id="2" name="Germany">
    <city id="21" name="Berlin"/>
    <city id="22" name="Bonn"/>
  </country>
  <country id="3" name="France">
    <city id="31" name="Paris"/>
    <city id="32" name="Sanary"/>
  </country>
  <country id="4" name="Italy">
    <city id="41" name="Roma"/>
    <city id="42" name="Milano"/>
  </country>
</world>
```

Figure 2: A Sample Document

possible instructions are `xsl:if`, a conditional without "else", `xsl:foreach` for iteration, `xsl:choose` for branching like in a switch statement, `xsl:value-of` to convert the result of an XPath expression to a string, and `xsl:apply-templates`. The latter recursively applies the template rules to all nodes in the result of the XPath expression given in its `select` attribute.

Processing Model Processing a template rule always takes place with respect to a *current node* and a *current node list*. At the beginning, the current node is the root node of the source document and the current node list only consists of this same node. Given a current node and a current node list, the template to be applied is selected. To guarantee that at least one template is applicable, XSLT defines default template rules (see Fig. 6). The template rule selection step is described in detail in the next section.

Given the template rule to be applied, its template is instantiated and produces a part of the result document. The processing model does not only process single nodes but instead processes lists of nodes (the current node list). The result of processing a list of nodes is the concatenation of the processing results of its members in order. A template rule then typically selects another list for processing. Hence evaluation continues recursively.

Including and Importing Stylesheets Selecting the correct template rule to be applied to a given current node is a tricky and costly procedure. Since we want to be complete and precise on this point, we must consider including and importing stylesheets. The `xsl:include` element

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSLT
  version="1.0">
<xsl:template match="/world">
  <world>
  <xsl:apply-templates
     select="country"/>
  </world>
</xsl:template>
<xsl:template match="country">
  <country>
  <name>
    <xsl:value-of select="@name"/>
  </name>
  <xsl:apply-templates select="city"/>
  </country>
</xsl:template>
<xsl:template match="city">
  <city>
    <xsl:value-of select="@name"/>
  </city>
</xsl:template>
</xsl:stylesheet>
```

Figure 3: A Sample Stylesheet

allows to include another stylesheet referenced in its `href` attribute. Processing replaces the `xsl:include` element by the contents of the referenced stylesheet.

The `xsl:import` element, which is only allowed at the beginning of the stylesheet, allows to import another stylesheet referenced by its `href` attribute. It is processed like the `xsl:include` element except that the template rules of the importing stylesheet take precedence over those in the imported stylesheet.

Possible `xsl:import` elements of `included` stylesheets are moved up in the including stylesheet to occur before the root but after the last possibly already present `xsl:import` element. Since the imported stylesheet may import other stylesheets, we are faced with an import hierarchy.

A template rule in the import hierarchy is defined to have lower import precedence than another template rule if in the import hierarchy it would be visited before that template rule in a post-order traversal of the import hierarchy.

3 Optimizing Template Rule Selection

3.1 When a Template Rule's Pattern Matches

Let ν be a current node and λ be a current node list. Let P be a pattern in the `match` attribute of an `xsl:template` element. Remember that P is an XPath expression. XPath expressions are evaluated with respect to a current node and a current node list.

XSLT specifies that P matches for the given ν and λ if and only if there is an ancestor node ν' of ν ($\nu = \nu'$ is allowed) such that ν is in the result obtained by evaluating P with current node ν' and current node list λ' consisting

```
<?xml version="1.0"?>
<world>
  <country>
    <name>Germany</name>
    <city>Berlin</city>
    <city>Bonn</city>
  </country>
  <country>
    <name>France</name>
    <city>Paris</city>
    <city>Sanary</city>
  </country>
  <country>
    <name>Italy</name>
    <city>Roma</city>
    <city>Milano</city>
  </country>
</world>
```

Figure 4: Result of Stylesheet Processing

1.	A pattern of the form 'name' or '@name' has the default priority 0.
2.	If a pattern is a node test, i. e. tests for element or attribute nodes, the default priority is -0.5.
3.	In other cases the default priority is 0.5.

Figure 5: Default Priorities

only of ν'.

A literal implementation of this test computes all the ancestors ν' of ν. For each ancestor ν' it then evaluates the XPath expression P and checks whether ν is in the result. If this is the case for at least one ancestor, P matches. Otherwise it does not. This procedure is very expensive. Consider, for example, the pattern "//*". Checking the pattern with the above procedure results in visiting every node in the document. That is why we optimize this check in Section 3.3.

3.2 Conflict Resolution, its Complexity, and the Template Sequence Ω

It is easily the case that several patterns of different template rules match. Hence, a conflict resolution scheme is needed. XSLT specifies the following conflict detection and resolution procedure. Let T be the set of *all matching template rules*. Then apply the following steps:

1. Eliminate all matching template rules that have lower

```
<xsl:template match = "*|/">
      <xsl:apply-templates/>
</xsl:template>

<xsl:template match = "text()|@*">
      <xsl:value-of select = "."/>
</xsl:template>
```

Figure 6: Default rules for element, root, text and attribute nodes

import precedence than rules with the highest import precedence found in T.

2. Eliminate all matching template rules with a priority lower than the highest priority found among template rules in T.

If a template rule's pattern contains multiple alternatives separated by '|', then it is treated as a set of template rules, one for each alternative.

XSLT further specifies that

- it is an error if this leaves more than one matching template rule.
- an XSLT processor may signal the error; if it does not signal the error, it must recover by choosing from among the matching template rules that are left the one that occurs last in the stylesheet.

Note that following this procedure literally requires checking *all* template rules for a match. If we decide not to signal an error but always recover, the following helps us saving half of this enormous effort.

1. For all templates with no priority given, add the default priority.
2. Determine the highest and the lowest priority (P_H and P_L) of all template rules in the stylesheet and all directly or indirectly included or imported stylesheets.

   Define $\delta_H = P_H + 1$ and $\delta_L = P_L - 1$.
3. Recursively replace `xsl:include` elements by the template rules in the included stylesheet.
4. Call `resolve-import` (stylesheet, 0, δ_H) (see Fig. 8).
5. Add the default rules with priority δ_L.
6. Apply a stable sort to order template rules by ascending priorities.
7. Reverse the resulting sequence.

Figure 7: Computing the template sequence Ω

```
number resolve-import (stylesheet s , number w, number δ) {
    for each import element in document order
    let s' be the referenced stylesheet
    call w = resolve(s', w, δ)
    w += δ;
    for all templates T in s
            add w to the priority of T.
    replace all import elements in s by the sequence of
        template rules found in the referenced stylesheet
    return w
}
```

Figure 8: Procedure resolve-import

First, we define the *template rule sequence* Ω of a stylesheet such that

1. Ω contains all template rules of the stylesheet and the directly or indirectly included or imported stylesheets.
2. for any given document, current node, and current node list, the first template rule in Ω that matches is exactly the one the conflict resolution strategy of XSLT chooses (no matter whether there is a conflict or not).

Now we can iterate over Ω until we find the first matching template rule. Fig. 7 gives an algorithm to effectively compute Ω. For this algorithm we have:

Theorem *The algorithm given in Fig. 7 correctly computes Ω.*

For space restrictions we cannot give the proof of our theorems here (see [15]). Note that in the algorithm we cannot sort descending in step 6 and leave out step 7 since the 'physical' position of a template rule in a stylesheet influences rule selection.

Ω can be computed once for a stylesheet and used for every application of it, i. e. Ω can be computed at compile time. Although using Ω already saves half of the effort on average, we further optimize the template selection process in Section 3.4.

The question remains what happens if we want to signal an error or warn the user of conflicting template rules. Obviously, we can do so at runtime at the according costs. What about compile time? For any two template rules with the same priority, we have to check whether there is a document and a current node in the document such that both their patterns match. Unfortunately, this is undecidable:

Theorem *For two given template rules it is undecidable whether they conflict or not.*

This theorem is derived from the following theorem which is interesting in itself.

Theorem *Given two XPath expressions P_1 and P_2. It is undecidable whether there exist a document D and a node ν in D such that the results of evaluating P_1 and P_2 with current node ν and current node list $< \nu >$ have an empty intersection.*

This theorem follows from the following theorem (use P_2 = '//*'):

Theorem *Given an XPath expression P, it is undecidable whether there exist a document and a node ν in D such that evaluating P with current node ν results in the empty set.*

For this theorem we have several proofs (see [15]). Each one uses only a small subset of the functionality of XPath. The first proof needs the child axis, some arithmetic operations (+, *, -), equality and integer constants. It then applies the solution of Hilbert's 10th problem [9]. The main idea is to use polynomials within predicates. We think that this part of the proof maybe of little relevance for practical stylesheets. The other proofs reduce Post's correspondence problem [16]. The first of these uses the `child` axis, equality, disjunction, conjunction, and the `nconc` function that concatenates all the string values of a set of nodes. The

second uses the `child` and `next-sibling` axis, equality, disjunction, conjunction and the `conc` function that concatenates two strings. It further uses the `position()` and `last()` function as well as `count()`. Alternatively to `next-sibling`, `prev-sibling` can be used, as our last proof showed. We do not know whether these scenarios capture realistic stylesheets.

The quintessence of these theorems is that signaling an error on conflicts is very expensive since in general it requires checking all template rules for a match. Since the goal of the paper is fast stylesheet processing, we choose to follow the recovery option of the XSLT specification and do not signal errors due to conflicts. Instead, we start with Ω and its related evaluation strategy and then apply further optimizations. The next subsection optimizes match checks for single template rules, the last subsection optimizes the overall template selection process.

3.3 Optimizing Single Match Checks

Let ν be a current node for which the applicability of a template rule T is to be checked. In order to do so, we have to evaluate T's pattern P for all anchestors ν' of ν, including ν itself. If we denote by $eval^{fw}_{\nu'}(P)$ the result of evaluating the Xpath expression P with current node ν' and current node list $< \nu' >$, we have to check whether for all $\nu' \quad \nu \in eval^{fw}_{\nu'}(P)$. In case of a success we can stop the evaluation early, otherwise we have to go all the way through!

If P contains only a single '//', this potentially results in the traversal of the whole document. Even if no '//' is present, larger than necessary parts of the document are traversed as illustrated in Fig. 9. The parts traversed for checking the pattern '/a/b/c' are indicated by a dotted line: it is the whole tree.

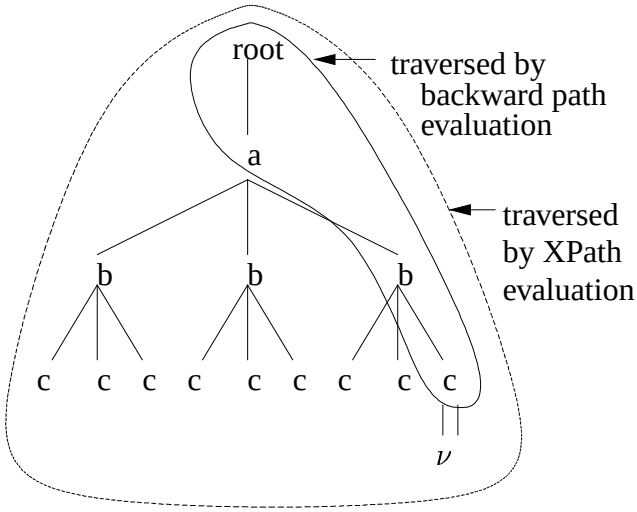


Figure 9: Traversed nodes for checking the pattern '/a/b/c' with current node ν

Intuitively, it is cheaper to start the test at node ν and check whether it is a c-node. Then move up to the parent and check whether it is a b-node and so on. The nodes traversed by this *backward* evaluation are also indicated in Fig. 9.

The idea of evaluating a given path expression backwards is not new. It appears for example in the object-oriented context (e.g. [4]) and in the semi-structured context (e.g. [14]). The basic idea of any of these approaches is to optimize the computation of the result of a path expression. As an invariant, it is always true that the *result* obtained by backward evaluation is the *same* as the *result* obtained by forward evaluation. The difference between forward and backward evaluation lies solely in the realm of efficiency. In contrast, we take an approach where we translate an XPath expression P (from now on synonymously called forward path) into a backward path P'. In all but trivial cases, the evaluations of P and P' will *differ*.

Let β be the translation process of translating a forward path into a backward path (described below). Let the backward path P' be the result of $\beta(P)$ for some forward path P. If we denote the evaluation of a backward path P' with current node ν by $eval^{bw}_{\nu}(P')$, then the following holds: $(\exists \nu' \in \text{ancestor-or-self}(\nu)$:

$$\nu \in eval^{fw}_{\nu'}(P)) \Longleftrightarrow (eval^{bw}_{\nu}(P') \neq \emptyset).$$

The backward path then captures our intuition in that its evaluation requires only a small fraction of the document to be traversed (as in Fig. 9). Hence, we replace the standard evaluation of the right-hand side by the much cheaper evaluation of the left-hand side. Since we are not interested in the result of the backward evaluation but only in the fact whether the result is empty or not, the above equivalence gives us the correctness of our approach. In the remainder of this subsection we formalize backward paths, their evaluation, and β. The formalization of backward paths will be needed in the next subsection.

As in XPath expressions, backward paths are allowed to carry predicates for nodes. We borrow XPath's syntax and emphasize the existence of the base predicates `root()`, `element()` and `attribute()` as well as the function `name()`. The `root()` predicate evaluates to true if the current node ν is the root node of the document that contains ν. The predicates `element()` and `attribute()` evaluate to true if the current node is an XML element or attribute node, respectively. The function `name()` returns (as in XPath) the name of an element or attribute node.

We can define *backward paths* inductively:

- $.[p]$ is a backward path and
- $P\backslash.[p]$ and $P\backslash\backslash.[p]$ are backward paths for any backward path P.

The predicate $[p]$ is optional. '$\backslash$' is called *parent* and '$\backslash\backslash$' *ancestor-or-self axis*[1]. The dot('.') is—strictly spoken—

[1] XSLT also allows '..', the parent axis which can easily be reflected by adding a child axis '..' to backward paths.

not needed but makes a nice contribution to the graphical representation of backward paths.
An example of a backward path is

.[element() ∧ name()='c']
\.[element() ∧ name() = 'b']
\.[element() ∧ name()] \.[root()].

This backward path happens to be $\beta(/a/b/c)$.

The challenge to make backward paths work correctly lies in some nitty gritty details. For example, a document's root node is an additional node atop the document node (the top node of the contents of the document in XML terminology). Another detail is the provision to always provide a context that allows the correct evaluation of the XPath functions `position()` and `last()`. Hence, the semantics might look a little more complex than expected.

We define

$$\text{eval}_\nu^{bw}(.[p]) = \{\nu | \nu \in \text{eval}_\nu^{fw}(.[p])\}$$

$$\begin{aligned}\text{eval}_\nu^{bw}(P\backslash.[p]) = \{\nu' | \exists \quad & \nu'' \in \text{eval}_\nu^{bw}(P) \wedge \\ & \nu' \in parent(\nu'') \wedge \\ & \nu \in \text{eval}_{\nu'}^{fw}(./*[p])\}\end{aligned}$$

$$\begin{aligned}\text{eval}_\nu^{bw}(P\backslash\backslash.[p]) = \{\nu' | \exists \quad & \nu'' \in \text{eval}_\nu^{bw}(P) \wedge \\ & \nu' \in \text{ancestor-or-self}(\nu'') \wedge \\ & \nu \in \text{eval}_{\nu'}^{fw}(.//*[p])\}\end{aligned}$$

Note that if a predicate p does not contain `position()` or `last()`, then p can always be evaluated locally at the current node without going up to a parent or ancestor and down to the children or descendants again. If p contains at least one of these functions and p occurs in conjunction with '\', then siblings of the node under test have to be traversed. Only in the remaining case, where p contains one of the above functions and occurs in conjunction with '\\', larger parts of the document have to be traversed.

We define the translation β of a forward path into a backward path as follows:

$$\beta(P) = \begin{cases} \beta'(P')\backslash.[\text{root}()] & \text{if P = / P'} \\ \beta'(P')\backslash\backslash.[\text{root}()] & \text{if P = // P'} \\ \beta'(P)\backslash\backslash. & \text{otherwise} \end{cases}$$

with

$$\beta'(P) = \begin{cases} \beta'(P')\backslash\alpha(N) & \text{if P = N/P'} \\ \beta'(P')\backslash\backslash\alpha(N) & \text{if P = N//P'} \end{cases}$$

where N is a node specifier matching α's argument patterns in

$$\begin{aligned} \alpha(a[p]) &= .[\text{ element() } \wedge \text{ name()='a' } \wedge p] \\ \alpha(*[p]) &= .[\text{ element() } \wedge p] \\ \alpha(@a[p]) &= .[\text{ attribute() } \wedge \text{ name()='a' } \wedge p] \\ \alpha(@*[p]) &= .[\text{ attribute() } \wedge p] \end{aligned}$$

Again, the predicates are optional. The result of $\beta(/a/b/c)$ has already been provided above. For more examples, see the next section.

3.4 Optimizing the Overall Selection Process

Optimizing the costs of evaluating a bunch of conditions to make a decision or determining a choice is often achieved by using and optimizing decision trees. In a typical decision tree, every node is labelled by a predicate and for every non-leaf node there exist two outgoing edges for the positive and the negative case. Decision trees of this form are not useful in our context: we are talking about nodes, node sets and occurrence and need to embed the notion of axis traversal into the decision tree. Hence, we introduce the *axis enhanced decision tree*. It is defined as follows. An *axis enhanced decision tree* (AEDT) is an ordered rooted tree whose nodes are labeled with predicates of the same form as in backward paths and whose edges are labeled by either `self`, '\', or '\\'. Additionally, leaf nodes contain references to template rules. Apart from the last point, every backward path is an axis enhanced decision tree where its leftmost node (dot) becomes the root. AEDT's are evaluated in a strict left to right order. Evaluation for a given current node starts at the root of the AEDT. For any given AEDT node n, its predicate is evaluated for the current node. If it evaluates to true, and n is a leaf node, its template rule is selected. Otherwise we follow the outgoing edges in left to right order. For every edge, we evaluate the path expression associated with it. If the label is `self`, then the current node remains unchanged. If the label is '\', the parent of the current node becomes the new current node. In case of '\\', every ancestor of the current node becomes the new current node. For each new current node, we evaluate the predicate of the node the edge under consideration leads to. If n's predicate evaluates to false, we proceed with the next unprocessed edge of the AEDT.

The template sequence Ω can now be translated into an *initial AEDT* by translating each pattern into a backward path and then into an AEDT. All AEDT's are then collected under a new root and the edges from the root to the pattern's AEDTs are labelled by `self`. The resulting AEDT for the template rules in Fig. 3 is given in Fig. 11 a). Since the default rules will never fire for our example, we omitted them as we will in the rest of the paper.

So far, not much has been gained over the simple template rule selection procedure using Ω. However, the AEDT gives us a general representation of all tests to be performed and allows subsequent optimizations.

Before giving our optimization method for AETDs, we present two theorems illuminating the complexity of optimizing AETDs.

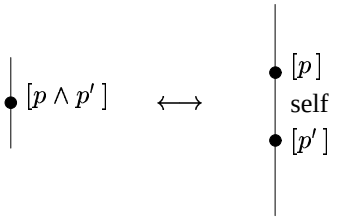
Theorem *Deciding whether two AEDTs are equivalent is undecidable.*

Equivalent here means that the same leaf node is selected for any current node in any document. Again, we refer to [15] for full proofs.

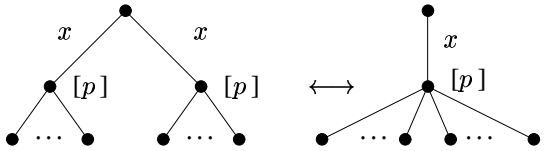
Theorem *Under any reasonable cost function it is undecidable whether a given AEDT is optimal even if we know that it is equivalent to the initial AEDT of some template sequence* Ω.

A cost function is reasonable if it adds positive costs for all

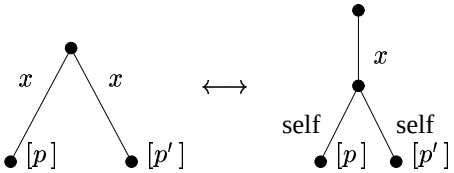
predicates occurring in the AEDT and sums up these costs. The proofs build on the fact that there exist predicates that cannot be eliminated safely but possibly.



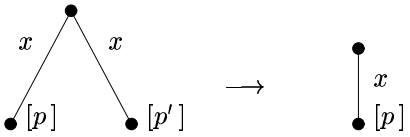
Predicate Extraction
(p' does not contain `position()` or `last()`)



Node Factorization ($x \in \{'\text{self}', '\backslash'\}$)



Edge Factorization



Simplification ($x \in' \{\text{self}', '\backslash'\}$ and $p' \Rightarrow p$)

Figure 10: AEDT Transformation

These theorems show that optimization of the initial AEDT should take place in small, carefully chosen steps. We use the rewrite rules contained in Fig. 10 to transform the initial AEDT into an optimized form. In general, the rules are applied in a left to right manner. Some rules rely on additional preconditions. The goal of applying these rules is to eliminate as many redundant predicate evaluations and traversals as possible. In order to do so, we perform the following steps to any initial AEDT:

1. Apply predicate extraction for all `element()` and `attribute()` predicates.
2. Apply node factorization on nodes containing only the `element()` or `attribute()` predicate.
3. Apply steps 1 and 2 for predicates involving the `name()` function.
4. Apply as many edge and node factorization as well as simplification steps as possible. Use predicate extraction to enable these rules.

An initial AEDT and its optimized form derived by applying the above algorithm is given in Fig. 11 a) and b). Evaluation of the AEDT can then be optimized locally at the node level. Consider for example the case where all child nodes of a given node n have the form `name() = const`. If all the edges from n to these children are labeled by `self` or '\', then we do not need sequential checking but can replace this step by a more efficient jump table.

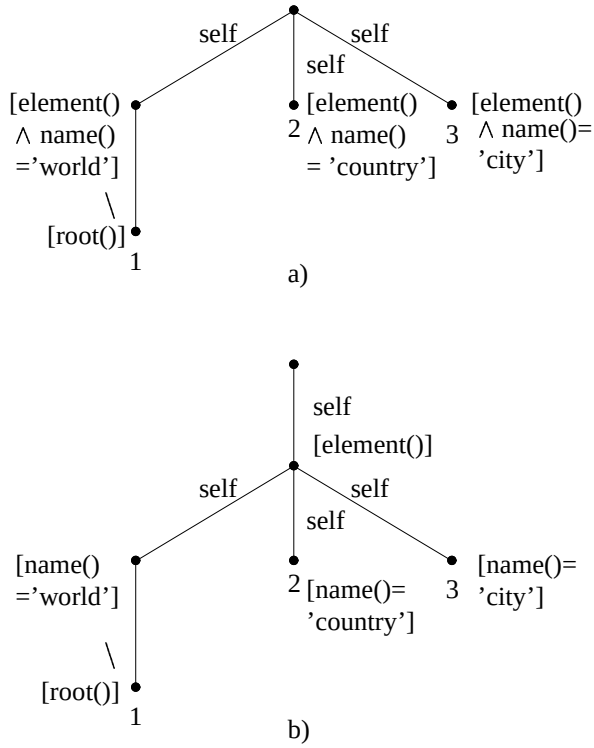


Figure 11: Initial AEDT and its optimized equivalent

4 From Stylesheets to Algebraic Expressions

4.1 Data Model, Algebra and the Rough Picture

Since order plays a crucial role in XSLT, our algebra works on sequences of tuples. For a logical relational algebra this is not really common. However, for a physical algebra this is a given fact even in the standard relational case since algebraic operators are implemented by applying the iterator principle [13]. We will describe our algebra also in terms of iterator implementations. Before we do so, let us clarify what our tuples look like.

First, we view a tuple as a sequence of attribute values. Each value can be a number or a string, but also a document node or even a pointer to an algebraic operator. The latter is used for routing the data flow as explained below. It is convenient for us if some attributes of a tuple are predefined. These are `streamId`, `resumeOp`, `current` (holding the current node) and `position` (holding the position of the current node). The resulting schema of a tuple is shown in Fig. 12. The first two attributes will be explained below.

The algebra now works on sequences of these tuples. The core of the algebra consists of the following operators: `DocScan`, `Sink`, `Map`, `Select`, `UnnestMap`, `Distributor`, `GPush` and `Collector`. All these operators are iterator-based. Additionally, there is a `stack` operator with the usual `top`, `push` and `pop` functions. It is needed to resolve recursion. The next section will show how it can be eliminated during the optimization of the initial plan. Let us now explain the remaining algebraic operators. It might help if the reader already glances at the initial plan (Fig. 13) for our sample stylesheet (Fig. 3).

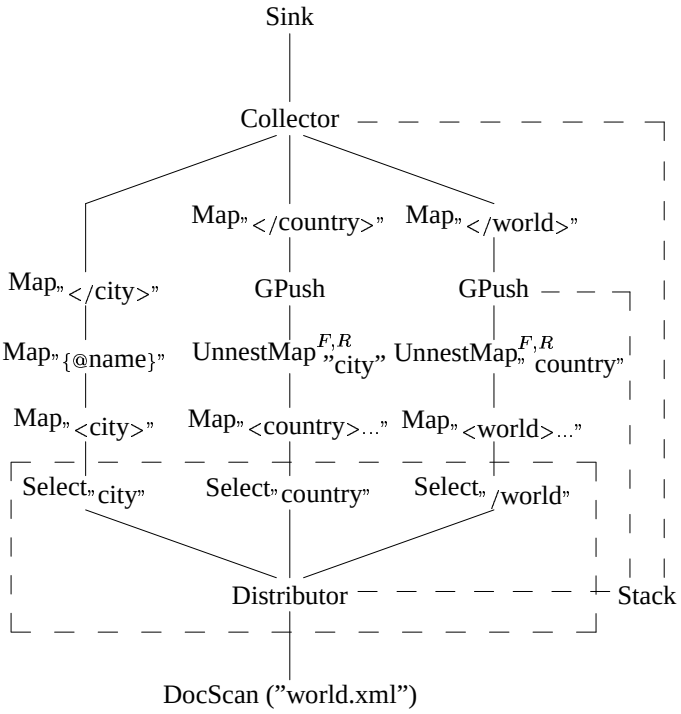


Figure 13: Initial Plan

The `DocScan` operator creates a single tuple for every document to be processed by the current stylesheet. It initializes the `current` attribute with the root node of the document. Independent of how the documents are stored, this is an easy task. If they are stored in a relational database, selecting root nodes of documents is easy in any known encoding, for example [12, 20, 18, 19, 21]. If the documents are stored in a native XML database, retrieving document root nodes is also an easy task, as for example in Natix [11]. The `Sink` operator does nothing but eating all its input tuples. The `Map` operator applies a function to a tuple. In our case this will always be a print function[2]. We will give the text it prints as its subscript. This text may contain XPath expressions whose result is to be printed in textual form. We chose '{' and '}' to escape these XPath expressions. The `Select` operator applies a predicate to its input tuple and—this is different from standard select—signals an end of stream if the predicate evaluates to false. The `Distributor` together with the immediately following `Select` operators implements the template selection process.

If we glance at Fig. 13 again, we see the `Select`s and the `Distributor` surrounded by a box. This indicates that they should be implemented *together* as a single operator using the AEDT approach of the previous section. However, we keep them separate until the end of the optimization process since we will need explicit selections in the next section. The predicates of the selection operators are given as the XPath expressions that occur in the `match` attributes of the template rules. The semantics of these predicates is the one described in Sec. 3.1.

In essence, the distributor sends each tuple to its output streams (in order!) until one of them is able to process it successfully. Since it is standard to implement algebraic operators as iterators [13], we see a misfit. This is remedied by the `Collector`. It simply asks the `Distributor` on which stream the next `next` call is to be issued. Summarizing, a `Distributor`-`Collector` pair together with the `Selection`s implements the template selection process. Every stream in between then is the result of translating a single template rule's body into the algebra. We do not specify the output of the `Collector` here since it does not matter because of the `Sink` on top of the plan. However, the output will matter and be specified precisely in the next section.

`UnnestMap` takes a path expression and returns for every node in the result of the path expression a tuple with its `current` attribute set to that node. It can be parameterized in two aspects. First, it can copy the argument tuple it got from a `next` call as its first output tuple before all the other tuples are generated by evaluating the path expression. This option is indicated by an 'F' in the superscript. If the original input tuple is to be copied to the output after the nodes resulting from the evaluation of the path expression, we denote this by 'L'. Second, `UnnestMap` is able to reverse the result order. XPath specifies that nodes in the result set are ordered in document order. If we add an 'R' to `UnnestMap`'s superscript, it generates the result set in reverse document order. Note that this does not imply buffering and reversing the result set, it can be implemented by merely reversing traversal operations. For example, when following the child axis, we produce the children starting with the last child instead of the first. As an example consider $\texttt{UnnestMap}^{F}_{country}$ applied to the node with `id` 1

[2]This simplifies the exposition only. It is straightforward to add functions that construct a tree of throw SAX events.

stream id	resume op	current	position	... other attributes ...

Figure 12: Tuple Scheme

(see Fig. 2). It produces 1, 2, 3, 4. $\texttt{UnnestMap}^{F,R}_{country}$ produces 1, 4, 3, 2 and $\texttt{UnnestMap}^{L}_{country}$ produces 2, 3, 4, 1.

The last operator to be described is `GPush`. It is assumed that it receives an input as produced by $\texttt{UnnestMap}^{F,R}$. This input is grouped. The first tuple is the original tuple and then those generated from the path expression of $\texttt{UnnestMap}^{F,R}$ follow. With such a group `GPush` now does the following: it marks the top tuple on the stack, pushes all subsequent tuples of one such group onto the stack, and then signals an end of stream. `GPush` marks the first tuple by storing its own address/id in the `resumeOp` attribute. This is necessary since every tuple must go through one complete stream between a `Distributor-Collector` pair. This is achieved in two phases. So far we have described the first phase where tuples are pushed onto the stack. They are eliminated from the stack only by the `Collector`, and eventually a group's first tuple reappears on the top of the stack. `GPush` then recognizes its responsibility by looking at the tuple's `resumeOp` attribute and passes it up the stream. It should be obvious by now that an `UnnestMap-GPush` pair is generated for every `xsl:apply-templates` instruction.

For details on the implementation of the `next` methods of the algebraic operators [15].

4.2 The Translation Process

There are two main problems when translating XSL stylesheets into an algebraic expression. The first one is the template selection process. This process has been discussed in the previous section. The second problem occurs with `xsl:apply-templates` instructions. These imply recursion of high complexity [5]. In principle, there should be two solutions to the problem: using a fixpoint operator as in Datalog [22] or using a stack to resolve recursion. We decided to take the second approach since order preserving fixpoint operations which are also efficient are not known to us. Maybe future research can help us here. Meanwhile, we use a `Stack` operator. Given these decisions and the algebra the algorithm for translating a stylesheet into the initial plan is defined as follows:

1. Compute Ω.
2. For every template rule in Ω create a subplan by
   - creating a single `Select` operator with the path of the pattern of the template rule as its predicate. This `Select` operator becomes the current plan.
   - For every instruction in the body of the template rule (in order of appearance):

     If the instruction is a string, create a `Map` operator with the string as its subscript and stack it onto the current plan.

     If the instruction is an `xsl:apply-templates` element with a path P in its `select` attribute, create an $\texttt{UnnestMap}^{F,R}_{P}$ operator and stack it onto the current plan. Then add a `GPush` to the current plan.

     (Other possible instructions exist in XSLT and their straightforward translations go here.)
3. Let a `Distributor-Collector` pair embrace all the plans of step 2 while obeying Ω's order.
4. Create a stack operator and make it known to all `Distributor`, `Collector` and `GPush` operators in the plan.
5. Add a `Sink` on top of the plan and a `DocScan` below it.

The plan created by steps 1-4 is called the stylesheet's *core plan*.

Theorem *The above translation is correct.*

Although we will not give the full proof by induction (see [15]) on the number of nodes to be processed, we give the main arguments of the proof because we think they will help the reader to understand what happens dynamically. An alternative is to follow the example of Fig. 1 through the initial plan. In doing so, it might be helpful to take a look at Fig. 14 which contains the stack status after every push or pop operation as well as the output produced by the plan.

The main arguments of the proof are:

1. The translation of the instructions other than `xsl:apply-template` is obviously correct.
2. Only two operators push tuples onto the stack: `Distributor` and `GPush`. Before each push, the tuple's `current` attribute contains a node that needs (further) processing. Other attributes (`resume`, `streamId`) are initialized correctly.
3. The order of streams in the `Distributor-Collector` pair directly reflects Ω's order.
4. Every node (= tuple's `current` content) whose template (= stream) is not yet decided upon (resume = 0, streamId = -1) is sent to every stream in order.
5. If a `select` fails for a tuple, this tuple never moves further upwards. That is, the `Collector`'s `next` call returns false. This implies that the next stream is tested for the same tuple.

```
Stack   [empty]
Stack   1
Out     <world>
Stack  1   4
Stack  1   4   3
Stack  1   4   3   2
Out   <country><name>Germany</name>
Stack  1   4   3   2  22
Stack  1   4   3   2  22  21
Out   <city>Berlin</city>
Stack 1   4   3   2  22
Out   <city>Bonn</city>
Stack  1   4   3   2
Out   </country>
Stack  1   4   3
Out   <country><name>France</name>
Stack  1   4   3  32
Stack  1   4   3  32  31
Out   <city>Paris</city>
Stack  1   4   3  32
Out   <city>Sanary</city>
Stack  1   4   3
Out   </country>
Stack  1   4
Out   <country><name>Italy</name>
Stack  1   4  42
Stack  1   4  42  41
Out   <city>Roma</city>
Stack  1   4  42
Out   <city>Milano</city>
Stack  1   4
Out   </country>
Stack  1
Out   </world>
Stack  [empty]
```

Figure 14: Stack dumps and output produced by the initial plan

6. If a `select` succeeds for a tuple T (which is always on top of the stack at this point in time), this tuple is processed by subsequent operators.

   If this is a `Map`, correctness is easily seen. The tuple is processed immediately and moves up to the next operator until it finally reaches the `Collector`, where it is removed from the stack.

   If there happens to be an `UnnestMap`-`GPush` pair along the stream, first the `UnnestMap` operator creates a group of tuples: the original tuple and those in the result of its path expression.

   Note that the original tuple is still on top of the stack. `GPush` correctly sets the `ResumeOp` attribute of this tuple to itself, indicating that it has been processed by all operators below it and needs to be processed by all operators above it. Then `GPush` pushes the other tuples onto the stack. The 'R' of the `UnnestMap` and the order-reversing effect of the stack cancel each other out.

   Let us assume that all tuples on the stack get processed and let us consider the moment the original tuple reappears on top of the stack. Since the `Collector` is the only operator that calls `pop` and atop of it is the `sink` operator, we can infer that the `Distributor` is asked for the next stream to issue a `next` call on.

   This stream is the one containing the `GPush` (determined by the `streamId` attribute) that pushed it and the `next`-calls ripple down until its `next` member function is called. `GPush` then looks at the top of the stack, recognizes itself in the `resumeOp` attribute and hands the tuple upwards.

Note that a plan may never halt. This is due to the fact that stylesheet processing itself may also not terminate. The problem to determine whether it halts or not is undecidable [5].

5 Optimizing the Initial Plan

Every node processed by a stylesheet enters the stylesheet's core plan at the `Distributor`. Except for the root node, `GPush` sends the nodes to the `Distributor` via the stack. The main idea of the optimization process is to successively replace `GPush` operations by the core plan and apply some simplifications until a good plan has been achieved or no more `GPush` operations are present. Consider our example plan in Fig. 13. Looking at the `DOCTYPE` (assumed to be metadata accessible to the template optimizer) of our example document (Fig. 2) we see that first the root node enters the core plan and the $\texttt{Select}_{/world}$ stream is selected. Hence, we start by replacing the `GPush` in this stream by the core plan. The result is shown in Fig. 15. Note that we left out the stack. A first subtlety occurs. We want every tuple that enters a `Distributor`-`Collector` subplan to leave it. Hence, we add a bypass stream to the inner `Distributor`-`Collector` subplan. Along this bypass, the entering tuple exits when all other nodes have been processed and it reappears on top of the stack. In order to make this work correctly, we mark the entering tuple and use an $\texttt{UnnestMap}^{L}$ in the outer plan. The 'R' has also been dropped since tuples are not pushed on the order reversing stack anymore. Another subtlety is the tuple structure which has to be changed. We need the predefined attributes for every `Distributor`-`Collector` pair. Apart from these two subtleties, there is no problem with replacing any `GPush` by a stylesheet's core plan.

After any such replacement we perform some optimizations. Consider again the plan in Fig. 15. `DocScan` produces a tuple whose `current` attribute is set to the root node of the document. Hence, we know that the only branch that qualifies for this tuple is the rightmost branch of the outer `Distributor`. We can eliminate the other two branches. This leaves the outer `Distributor`-`Collector` pair with only one stream in between. The

next optimization is to remove the `Distributor` and replace the `Collector` by a `Group` operator. `Group` reverses the effects of an $\texttt{UnnestMap}^L$. It only outputs the original tuple, i.e. the last in any group. The other tuples are ignored. Hence, `Group` can be implemented very efficiently.

We are now ready to pick another `GPush` for replacement by the core plan. We chose the `GPush` of the inner plan that lies on the $\texttt{Select}_{country}$ branch since looking at the DTD (Fig. 1) tells us that only `country` elements are children of `world` elements. We know (DTD) that only `city` elements are below `country` elements. Hence, all other branches can be eliminated from the innermost core plan. Again, this leaves a core plan with only one branch. We can replace the `Collector` by a group and eliminate the `Distributor`.

After these replacements we are left with a sequence of operations with no more `GPush`, `Distributor`, or `Collector` nodes. Hence, we can also leave out the stack operator. In general, whenever there is a non-recursive DTD, we can finally eliminate the stack by following the above procedure. But we are not quite done. We can eliminate the `Select` operations since we are sure the condition will evaluate to true. The resulting plan is shown in Fig. 15.

Last but not least, if there exist remaining `Distributor` and `Select` operations, they should be replaced by a powerful distributor implementation that uses the AEDT approach.

6 Conclusion

We started by investigating the template rule selection process. It turned out that its optimization is a hard problem: conflict detection at compile time and constructing optimal AEDTs are both undecidable problems. Then we saw that by small, carefully selected rewrite steps substantial optimization of a given AEDT is still possible. Next, we introduced an algebra and a translation process of a stylesheet into the algebra. Its subsequent optimization could not only lead to more efficient plans but also eliminate recursion in case of non-recursive DTDs.

Despite these achievements, there is a need for more research. Future research should try to find more optimization possibilities for AEDTs. A large area for future research—not even touched in this paper—is the combined optimization of queries constructing documents and XSL stylesheets processing them. Last but not least, there should be investigations on alternative ways to incorporate stylesheet processing into database engines.

Acknowledgement. I thank Simone Seeger for her help in preparing the manuscript and Thorsten Fiebig for many fruitful discussions.

References

[1] DB2 XML Extender. http://www-4.ibm.com/-software/data/db2/extenders/xmlext/.

[2] A survey of microsoft sql server 2000 xml features. http://msdn.microsoft.com/library/-default.asp?url=/library/en-us/dnexxml/html/-xml07162001.asp.

[3] XML, XSLT and Oracle8*i*. http://technet.oracle.com/sample_code/tech/xml/-xsql_servlet/sample_code_index.htm.

[4] J. Banerjee, W. Kim, and K.-C. Kim. Queries in object-oriented databases. In *Proc. IEEE Conference on Data Engineering*, pages 31–38, 1988.

[5] G. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Computational Logic*, pages 1137–1151, 2000.

[6] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. Technical report, World Wide Web Consortium, 1998. W3C Recommendation 10-Feb-98.

[7] J. Clark. XSL transformations (XSLT) version 1.0. Technical report, World Wide Web Consortium, 1999. W3C Recommendation 16 Nov. 1999.

[8] J. Clark and S. DeRose. XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium, 1999. W3C Recommendation 16 Nov. 1999.

[9] M. Davis, Y. Matijasevich, and J. Robinson. Hilbert's tenth problem. diophantine equations: positive aspects of a negative solution. In *Proc. of the Symp. on Hilbert Problems*, pages 323–378. American Math. Soc., 1976.

[10] S. Deach. Extensible stylesheet language (XSL) specification. Technical report, World Wide Web Consortium, 2001. W3C Recommendation 15 Oct 2001.

[11] T. Fiebig, S. Helmer, C.-C. Kanne, J. Mildenberger, G. Moerkotte, R. Schiele, and T. Westmann. Anatomy of a native xml base management system. Technical Report 01, University of Mannheim, 2002.

[12] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[13] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), June 1993.

[14] J. McHugh and J. Widom. Query optimization for XML. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 315–326, 1999.

[15] G. Moerkotte. Incorporating XSL processing into database engines. Technical Report 7, University of Mannheim, 2002.

[16] E. Post. A variant of a recursively unsolvable problem. *Bull. AMS*, 52:264–268, 1946.

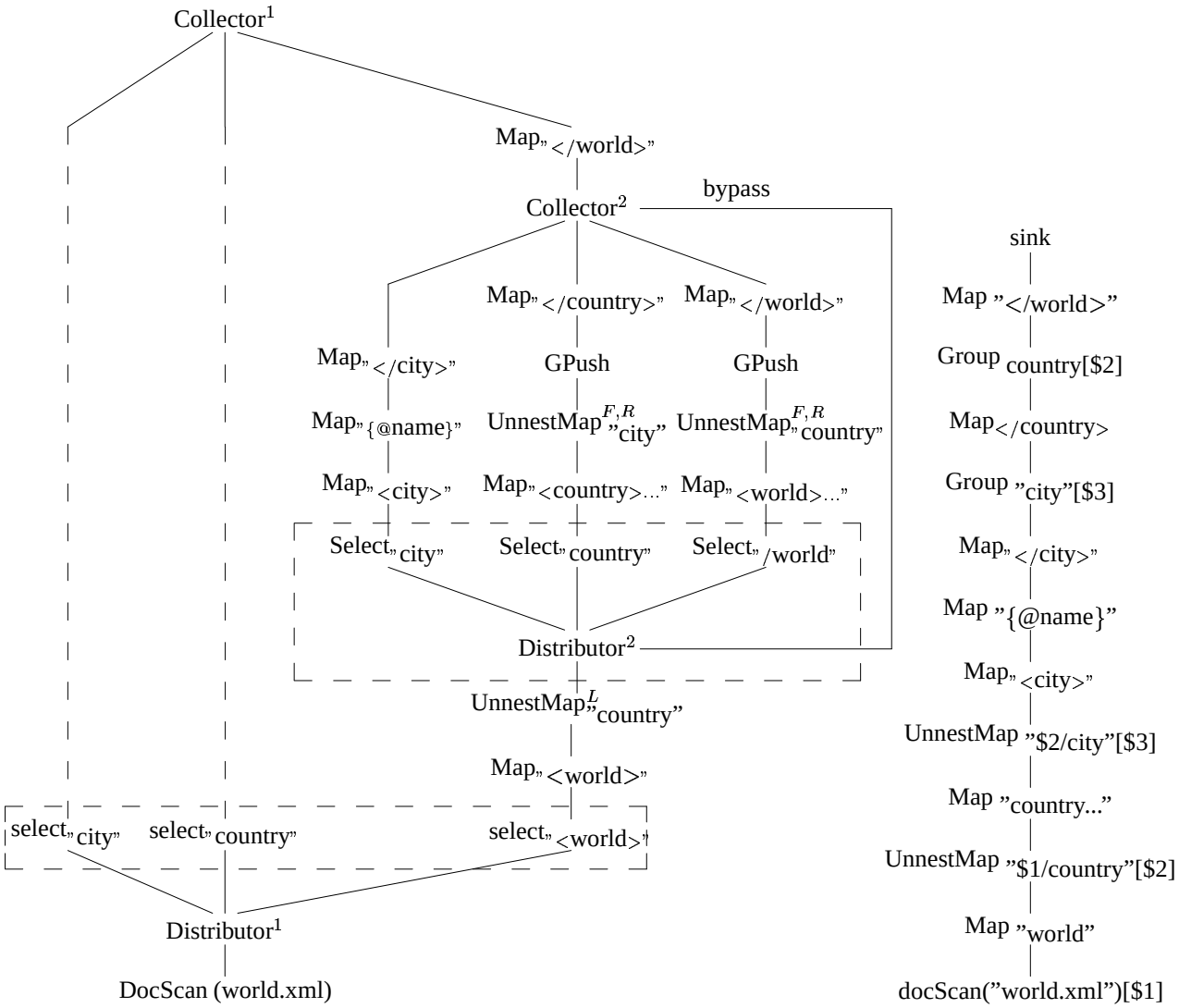


Figure 15: Result of GPush-Replacement and Final Plan

[17] M. Rys. Bringing the Internet to Your Database: Using SQLServer 2000 and XML to Build Loosely-Coupled Systems. In *Proc. IEEE Conference on Data Engineering*, pages 465–472, 2001.

[18] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2000.

[19] J. Shanmugasundaram, R. Barr E. J. Shekita, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 65–76, 2000.

[20] J. Shanmugasundaram, H. Gang, K. Tufte, C. Yhang, D. J. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 302–314, 1999.

[21] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2002. to appear.

[22] J.D. Ullman. *Database and Knowledge Base Systems*. Computer Science Press, 1989.