A Study on XSLT Transformation Method for Distributed XML

Mizumoto Hiroki

Graduate School of Library, Information and Media Studies
University of Tsukuba

March 2015

Contents

Chapter 1

Introduction

XML has been a de-fact standard format on the Web, and the sizes of XML documents have rapidly been increasing. *Distributed XML*[5, 2, 1, 6] is a novel form of XML document, in which an XML document is partitioned into fragments and managed separately in plural sites. Figures 1.1 and 1.2 show a simple example of a distributed XML document of multinational corporation clientele. In this example, one XML document is partitioned into four fragments f_1, f_2, f_3, and f_4, and f_1 is stored in site S_1, f_2 is stored in site S_2, and so on. Due to geographical and/or administrative factors, distributed XML is much suitable for managing some kind of XML documents, e.g., an XML document containing some separable subcontents that should be managed by different admins[1].

In this thesis, we consider XSLT transformation for distributed XML documents. A conventional approach for performing an XSLT transformation on a distributed XML document is to send all fragments to a specific site, then merge all the fragments into one XML document, and perform an XSLT transformation on the merged document. However, this "centralized" approach is inefficient due to the following reasons. First, in this approach an XSLT transformation processing is not load-balanced. Second, an XSLT transformation becomes inefficient if the size of the target XML document is large[17]. This implies that the centralized approach is inefficient even if the size of each XML fragment is small, whenever the merged document is large. In this pater, we propose a method for performing XSLT transformation efficiently for distributed XML documents. Our basic strategy is to transform each fragment at the site storing the fragment, send the transformed fragments to a specific site, and merge all the transformed fragments on the specific site. To achieve this strategy, however, we have a problem attributed to XSLT pattern. In our data model, a location path can be used as an XSLT pattern instead of a single label. Due to this, a site has to access other sites many times to evaluate an XSLT pattern, which causes a serious performance problem. Let S be a site, f be the fragment in S, and *pat* be an XSLT pattern, and consider checking if a node v in f matches *pat*. We need to find the ancestors v' of v (and some descendants of v') such that v is reachable from v' via *pat*. Since v' is often a node outside S, many accesses to sites outside S are required to check if v matches *pat*. For example, consider the XML fragments in Figs. 1.1 and 1.2, and suppose that we have the following XSLT template.

<xsl:template match="branches/branch[currency]//deal">

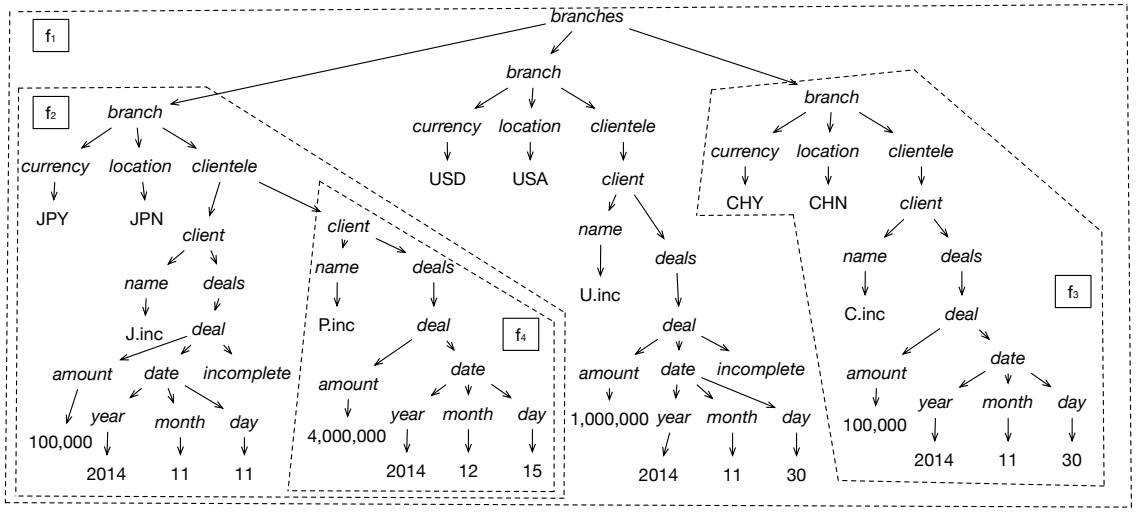

Figure 1.1 Multinational corporation clientele

...
</xsl:template>

To check if the node labeled by "deal" in f_4 matches the above template, we need to access S_2 and S_1 from S_4. To reduce such accesses, we propose two novel techniques; (1) precomputation of ancestors and (2) cache for predicate evaluation. For (1), each site S having a fragment f precomputes a path from the root of f to the root of the input XML document, called *root path*. By using a root path, ancestors v' of a node v can be obtained efficiently even if v' is stored in a site different from the site having v. As for (2), each site maintains a *cache* that stores results of predicate evaluations. In an XML document, sibling nodes tend to have the same label. Therefore, if a pattern matching such sibling nodes accesses "outside" sites, a lot of similar communications between sites may occur to evaluate the pattern. For example, consider evaluating pattern "$\downarrow^*$::branch[$\downarrow$::currency]/$\downarrow^*$::deal" for five sibling nodes $v_8, \cdots, v_{12}$ (Fig. 1.3). Without cache, due to the predicate "$\downarrow^*$::currency" an access from f_4' to f_2' is required for each of the five siblings. By storing the results of such predicate evaluations in a cache we can reduce accessing "outside" sites when evaluating predicates. We implemented our method in Ruby and made evaluation experiments. The result suggests that our method is more efficient than the centralized approach.

To show how XSLT is applied to distributed XML, consider again the XML tree shown in Fig. 1.1. It is often preferable that such trees are decomposed into a number of fragments and are distributed over the Internet for geographical or administrative reasons. For example, a client may request that his data is stored in a site located in his country since the site and its data must obey the laws of the country where the site is located (e.g., USA Patriot Act). In this example, we assume that f_1 is stored in American site S_1, f_2 is stored in Japanese site S_2, and so on. Here, suppose that the manager of this corporation requests a list consisting of, for each currency, the sales amounts of "incomplete" deals. By using the XSLT stylesheet shown in Fig. 1.4, we can easily obtain the list of sales amounts associated with currency of incomplete deals (Fig. 1.5). Note

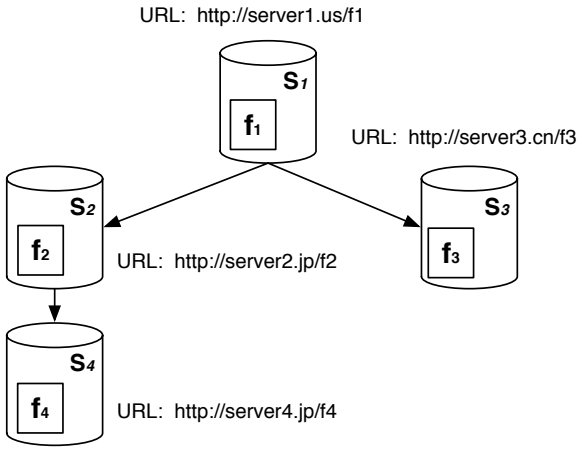

Figure 1.2 Four sites storing the fragments in Fig. 1.1

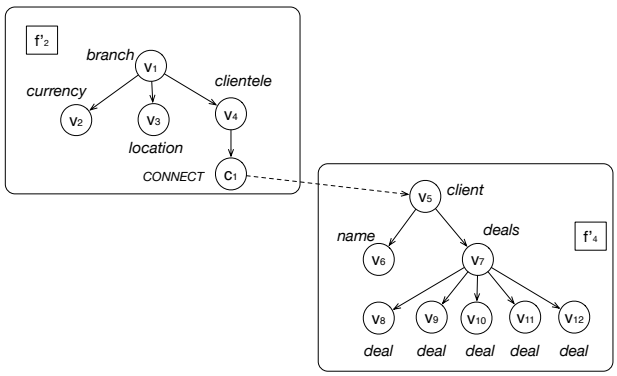

Figure 1.3 Sibling nodes labeled as same

that such a result cannot be obtained if we use XPath instead of XSLT.

Since XSLT is Turing complete[7], it is hard to plan a complete strategy of XSLT transformation for distributed XML. In this thesis, we focus on the top down transformation with node selection by using patterns, and thus use a top down tree transducer instead of the full XSLT. This tree transducer is an extended version of the unranked top-down tree transducer used in [11]. Our tree transducer is extended so that, in addition to a single label, a location path can be used as a match attribute of an XSLT template. Thus, this thesis adopts the "core" of XSLT transformation to focus on the distributed evaluation of XSLT pattern. However, our tree transducer can easily be extended so that it covers about a half of XSLT instructions/functions. Table 1.1 classifies the instructions/functions of XSLT 1.0 into two types A and B. Here, the instructions/functions f of type A can locally be calculated within the fragment in which f is evaluated, e.g., `xsl:text`. On the other hand, the instructions/functions f of type B may accesses several fragments beyond the fragment in which the f is evaluated, e.g., the select attribute of `xsl:for-each` may access outside of the fragment having the current node. Since the instructions/functions of Type A are not affected by how fragments are distributed, the instructions/functions can easily be incorporated into our tree transducer. Taking this into account, we believe that our tree transducer represents a practical class of XSLT transformation.

Table 1.1 XSLT 1.0 instructions and functions

	instruction	function
Type A	18	18
Type B	17	16
Total	35	34

```
<xsl:template match="branches">
  <data>
    <xsl:apply-templates/>
  </data>
</xsl:template>

<xsl:template match="branch">
  <branch>
    <xsl:apply-templates/>
  </branch>
</xsl:template>

<xsl:template match="currency">
  <currency>
    <xsl:apply-templates/>
  </currency>
</xsl:template>

<xsl:template match="deal[incomplete]/amount">
  <amount>
    <xsl:apply-templates/>
  </amount>
</xsl:template>
```

Figure 1.4 An example of XSLT stylesheet

```
<data>
  <branch>
    <currency>JPY</currency>
    <amount>100,000</amount>
  </branch>
  <branch>
    <currency>USD</currency>
    <amount>1,000,000</amount>
  </branch>
</data>
```

Figure 1.5 List of sales amounts with currency of incomplete deals

Related Work

A distribution design of XML documents is firstly proposed in [3]. There have been several studies on evaluations of XPath and other languages for distributed XML. [4, 5, 6, 8, 10] propose efficient XPath evalu-

ation algorithms for distributed XML. Given a tree t and an XPath query q, the algorithm in [4, 5, 6] traverses t and computes, for each node v in t, several vectors which records the evaluation values of subexpressions of q at v. The algorithm in [10] partitions an XML tree into fragments, selects appropriate fragments containing answers to the query, then performs a query processing on the fragments in parallel. [8] proposes, assuming that an XML tree is stored in relational tables, a scheme for parallel processing of XML tree using PC-clusters. [9] proposes a method for evaluating XQ, a subset of XPath, for vertically partitioned XML documents. [14] considers a regular path query evaluation in an distributed environment. [12] proposes a data-parallel approach for the processing of streaming XPath queries based on push down transducers. This approach permits XML data to be split into arbitrarily-sized chunks. [15] extensively studies the complexities of regular path query and structural recursion over distributed semistructured data. Besides query languages, [2] and [1] study on the complexities of schema design problems for distributed XML. To the best of the authors' knowledge, there is no study on XSLT evaluation for distributed XML.

Chapter 2

Definitions

Since our method is based on unranked top-down tree transducer, we first show related definitions. Let Σ be a set of labels. By T_Σ we mean the set of unranked Σ-trees. A tree whose root is labeled with $a \in \Sigma$ and has n subtrees $t_1, \cdots, t_n$ is denoted by $a(t_1 \cdots t_n)$. In the following, we always mean Σ-tree whenever we say tree. A *hedge* is a finite sequence of trees. The set of hedges is denoted by H_Σ. For a set Q, by $H_\Sigma(Q)$ we mean the set of Σ-hedges such that leaf nodes can be labeled with elements from Q. In the following, we use $t, t_1, t_2, \cdots$ to denote trees and $h, h_1, h_2, \cdots$ to denote hedges. We denote by $\lambda(u)$ the label of a node u. An *XSLT pattern* is specified as the match attribute value of an XSLT template. Formally, an *XSLT pattern* (*pattern* for short) is a subset of XPath location path defined as follows, where $\downarrow$ and $\downarrow^*$ denote child and descendant-or-self axes, respectively.

LocationPath ::= LocationStep | LocationPath '/' LocationStep
LocationStep ::= AxisName '::' NodeTest Predicate*
AxisName ::= $\downarrow$ | $\downarrow^*$
NodeTest ::= Any label in Σ
Predicate ::= '[' LocationPath ']'

From the above definition, a pattern pat can be expressed as $pat = ls_1/\cdots/ls_n$, where $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}]$, ax_i is an axis, l_i is a label, and $pd_{i,j}$ is a predicate. The *selection path* of pat, denoted $sel(pat)$, is the pattern obtained by dropping every predicate from pat, that is, $sel(pat) = ax_1 :: l_1/\cdots/ax_n :: l_n$.

Let t be a tree, $pat = ls_1/\cdots/ls_n$ be a pattern with $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}](1 \le i \le m)$, and v be a node of t. Suppose that we have an XSLT template whose match attribute value is pat. Then the XSLT template can be applied to v if there is an ancestor v' of v such that v is reachable from v' via pat. $M_{pat}(t, v, pat)$ denotes the set of such ancestors of v. Formally, $M_{pat}(t, v, pat)$ is defined as follows.

$$M_{pat}(t, v, pat) = \begin{cases} V(t, v, ls_n) & \text{if } n = 1, \\ \{v'' \mid v'' \in M_{pat}(t, v', pat'), v' \in V(t, v, ls_n)\} & \text{otherwise,} \end{cases}$$

where $pat' = ls_1/\cdots/ls_{n-1}$ and $V(t, v, ls_i)$ is the set of ancestors v' of v such that v is reachable from v' via ls_i,

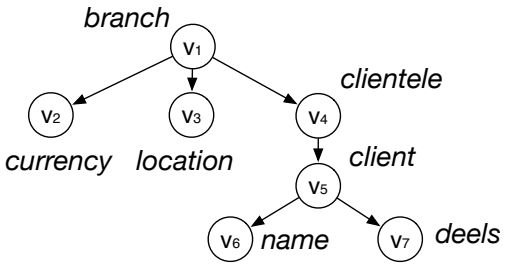

Figure 2.1 Tree t_e

defined as follows. First, if $i = 1$ (i.e., the leftmost location step), then

$$V(t, v, ls_1) = \begin{cases} \{v\} & \text{if } \lambda(v) = l_1 \text{ and } \bigwedge_{k=1,\cdots,m_1} M_{pd}(t, v, pd_{1,k}) \neq \emptyset, \\ \emptyset & \text{otherwise,} \end{cases}$$

where $M_{pd}(t, v, pd_{i,k})$ denotes the set of nodes reachable from v via predicate $pd_{i,k}$ in t (defined later). Thus, v satisfies $pd_{i,k}$ iff $M_{pd}(t, v, pd_{i,k}) \neq \emptyset$. Second, if $i > 1$, then

$$V(t, v, ls_i) = \begin{cases} \{v' \mid v' \text{ is a parent of } v \text{ in } t, \lambda(v) = l_i, \bigwedge_{k=1,\cdots,m_i} M_{pd}(t, v, pd_{i,k}) \neq \emptyset\} & \text{if } ax_i = \downarrow, \\ \{v' \mid v' \text{ is an ancestor of } v, \lambda(v) = l_i, \bigwedge_{k=1,\cdots,m_i} M_{pd}(t, v, pd_{i,k}) \neq \emptyset\} & \text{if } ax_i = \downarrow^* \end{cases}$$

where, ax_i is the axis of ls_i.

Then let us show the definition of $M_{pd}(t, v, pat)$.

$$M_{pd}(t, v, pat) = \begin{cases} V'(t, v, ls_1) & \text{if } n = 1, \\ \{v'' \mid v'' \in M_{pd}(t, v', pat''), v' \in V'(t, v, ls_1)\} & \text{otherwise,} \end{cases}$$

where $pat'' = ls_2/\cdots/ls_n$ and $V'(t, v, ls_i)$ denotes the set of nodes reachable from v via ls_i, that is,

$$V'(t, v, ls_i) = \begin{cases} \{v' \mid v' \text{ is a child of } v \text{ in } t, \lambda(v') = l_i, \bigwedge_{k=1,\cdots,m_i} M_{pd}(t, v', pd_{i,k}) \neq \emptyset\} & \text{if } ax_i = \downarrow, \\ \{v' \mid v' \text{ is a descendant of } v \text{ in } t, \lambda(v') = l_i, \bigwedge_{k=1,\cdots,m_i} M_{pd}(t, v', pd_{i,k}) \neq \emptyset\} & \text{if } ax_i = \downarrow^* \end{cases}$$

For example, let t_e be the tree shown in Fig. 2.1, $pat = \downarrow^*$::branch/$\downarrow$::currency and $pd = \downarrow^*$::clientele/$\downarrow^*$::deals. Then $M_{pat}(t_e, v_2, pat) = \{v_1\}$ and $M_{pd}(t_e, v_1, pd) = \{v_7\}$.

In this thesis, we use an extended version of the unranked tree transducer used in [11]. Formally, a *tree transducer* is a quadruple (Q, Σ, q_0, R), where Q is a finite set of *states*, $q_0 \in Q$ is the initial state, and R is a finite set of rules of the form $(q, pat) \rightarrow h$, where pat is a pattern, $q \in Q$ and $h \in H_\Sigma(Q)$ (in the original transducer[11], pat is restricted to a single label). A state corresponds to the mode attribute value of an XSLT template.

The translation defined by a tree transducer $Tr = (Q, \Sigma, q_0, R)$ on a tree t in state q, denoted by $Tr^q(t)$, is inductively defined as follows.

R1: If $t = \epsilon$, then $Tr^q(t) := \epsilon$.

R2: If $t = a(t_1 \cdots t_n)$ and there is a rule $(q, pat) \to h$ in R with $M_{pat}(t, a, pat) \neq \emptyset$ for some pattern pat, some $q \in Q$, and some $h \in H_\Sigma$, then $Tr^q(t)$ is obtained from h by replacing every node u in h labeled with $p \in Q$ by the hedge $Tr^p(t_1) \cdots Tr^p(t_n)$.

R3: If $M_{pat}(t, a, pat) = \emptyset$ for every pattern pat, every $q \in Q$, and every $h \in H_\Sigma, (q, pat) \to h$ in R, then $Tr^p(t) := \epsilon$.

The transformation of t by Tr, denoted by $Tr(t)$, is defined as $Tr^{q_0}(t)$.

Example 1 Let $Tr = (Q, \Sigma, p, R)$ be a tree transducer, where

$$
\begin{aligned}
Q &= \{p, q\}, \\
\Sigma &= \{\text{branch, currency, location, clientele, client, name, deals}, x, y, z\}, \\
R &= \{(p, \downarrow^*\!\text{::branch}) \to x(p\,q), (q, \downarrow^*\!\text{::currency}) \to z, \\
&\quad (p, \downarrow^*\!\text{::branch}[\downarrow\text{::location}]/\downarrow\text{::clientele}) \to y(p), \\
&\quad (p, \downarrow^*\!\text{::branch}/\downarrow^*\!\text{::client}) \to x(y)\}.
\end{aligned}
$$

Tr corresponds to the XSLT script shown in Fig. 2.2. For example, consider the rule $(p, \downarrow^*\text{::branch}) \to x(p\,q)$ in R. This corresponds to the first template in Fig. 2.2. The state p in the left-hand side of the rule corresponds to the mode attribute value of the template, and the pattern "$\downarrow^*$::branch" in the left-hand side of the rule corresponds to the match attribute value. Consider transforming the tree t_e shown in Fig. 2.1 by Tr. Since the initial state of Tr is p and the root v_1 of t_e is labeled by "branch", the first rule $(p, \downarrow^*\text{::branch}) \to x(p\,q)$ is applied to t_e and we obtain the tree shown in Fig. 2.3(1), where t_1 is the subtrees rooted at v_4 of t_e. Since there is no rule applicable to v_2 in state p, $Tr^p(v_2) = \emptyset$. Similarly $Tr^p(v_3) = \emptyset$, $Tr^q(v_3) = \emptyset$ and $Tr^q(t_1) = \emptyset$. Consider $Tr^p(t_1)$. Since the third rule $(p, \downarrow^*\text{::branch}[\downarrow\text{::location}]/\downarrow\text{::clientele}) \to y(p)$ can be applied to t_1, we obtain the tree shown in Fig. 2.3(2), where t_2 is the subtree of t_e rooted at v_5. Proceeding this transformation, we obtain $Tr(t_e)$ shown in Fig. 2.3(3).

In this thesis, we consider a setting in which an XML tree t is partitioned into a set F_t of disjoint subtrees of t, where each subtree is called *fragment*. For example, the XML tree $t \in T_\Sigma$ in Fig. 1.1 is partitioned into four fragments, f_1, f_2, f_3, f_4. We allow arbitrary "nesting" of fragments. Thus, fragments can appear at any level of the tree. For a tree t, the fragment containing the root node of t is called *root fragment*. In Fig. 1.2, the root fragment is f_1. Each fragment is stored in a *site*. The site having the root fragment is called *root site* and the other sites are called *slave sites*. For example, in Fig. 1.2 S_1 is the root site and S_2, S_3, S_4 are slave sites. We assume that no two fragments are stored in the same site.

For two fragments f_i and f_j, we say that f_j is a *child fragment* of f_i if the root node of f_j corresponds to a leaf node v of f_i. In order to represent a connection between f_i and f_j, we use a *connecting node* at the position of v which refers the root node of f_j. Every connecting node is labeled by "CONNECT" and has a url attribute that represents the URL of the site having f_j. For example, in Fig. 3.1 connecting node c_1 is inserted into $f_{e,1}$ at the position of v_5. If the fragment in site S has a child fragment stored in S', then S' is a *child site* of S (S is the *parent* site of S'). For example, in Fig. 1.2 S_1 has two child sites S_2 and S_3.

```
<xsl:template match="branch" mode="p">
  <x>
    <xsl:apply-templates mode="p" />
    <xsl:apply-templates mode="q" />
  </x>
</xsl:template>

<xsl:template match="currency" mode="q">
  <z>
    <xsl:apply-templates mode="q" />
  </z>
</xsl:template>

<xsl:template match="branch[location]/cli
 mode="p">
  <y>
    <xsl:apply-templates mode="p" />
  </y>
</xsl:template>

<xsl:template match="branch//client"
 mode="p">
  <x>
    <y />
  </x>
</xsl:template>
```

Figure 2.2 An example XSLT script

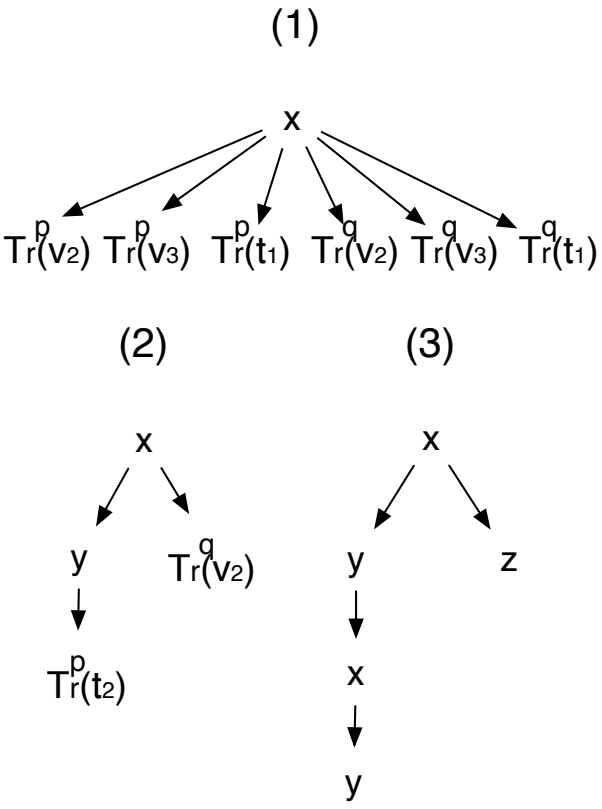

Figure 2.3 Tree transformation by Tr

Chapter 3

Transformation Method

In our transformation method, all the sites S transform the fragment f stored in S in parallel, in order to avoid transformation processes being centralized on a specific site. If the pattern of each transformation rule is a single element, it is rather easy to achieve this strategy; transform each fragment f at the site storing f, send all the transformed fragments to the root site, and merge all the transformed fragments on the root site. However, if a location path can be used as a pattern instead of a single label, a site has to access other sites to evaluate the pattern. Let S be a site, f be the fragment in S, and $pat = ls_1/\cdots/ls_n$ be a pattern, and consider checking if a node v in f matches pat. We need to do the following.

a) Find the ancestors v' of v such that v is reachable from v' via $sel(pat)$.
b) For each node v'' on the path from v' to v, check if v'' satisfies predicates of pat.

Both (a) and (b) may require accesses to sites outside S. In order to do (a) efficiently, S precomputes a path called *root path*, from the root node of f to the root node of the input tree. By using this root path, (a) can be done without accessing sites outside S. For (b), each site maintains a *cache* that stores, for nodes v' such that v is reachable from v' via $ls_i/\cdots/ls_n$ and predicates pd of ls_{i-1}, if v' satisfies pd. This can reduce accesses to sites outside S.

3.1 Master-XSLT and Slave-XSLT

We now present the details of our method. We first show two "main" XSLT processors Master-XSLT and Slave-XSLT. Master-XSLT is used in the root site and Slave-XSLT is used in the slave sites, as follows.

1. In the root site, Master-XSLT transforms the root fragment.
2. In each slave site S Slave-XSLT transforms the fragment in S and send the transformed result to the root site.
3. Master-XSLT merges (1) the transformed root fragment and (2) the transformed fragments received from the slave sites.

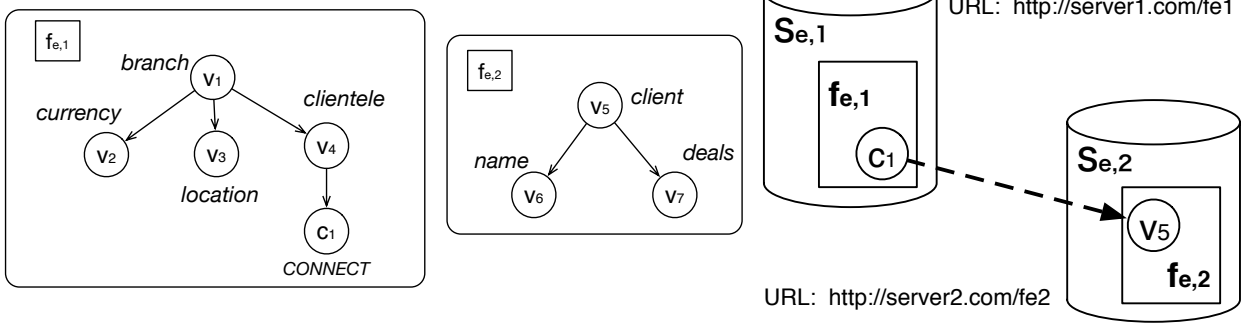

Figure 3.1 Fragments $f_{e,1}, f_{e,2}$ of t_e

Figure 3.2 Sites $S_{e,1}$ and $S_{e,2}$

For example, consider the fragments and the sites shown in Fig. 1.2. First, Master-XSLT transforms the root fragment f_1 to f_1' in the root site S_1. Second, in S_2 (resp., S_3 and S_4) Slave-XSLT transforms the fragment f_2 (resp., f_3 and f_4) and send the transformed result f_2' (resp., f_3' and f_4') to S_1. Among each transformation process, some communications between sites may occur by evaluating patterns. Finally, Master-XSLT merges the transformed fragment f_1' and the received fragments f_2', f_3' and f_4'.

To describe the "precomputation" of a root path, we need some definitions. Let $t \in T_\Sigma$ be a tree, F_t be the set of fragments obtained from t, $f \in F_t$ be a fragment, and v be a node of f. We use the following notation.

- $\mathrm{child}(f, v) = \{v' \mid v' \text{ is a child of } v \text{ in } f\}$
- $\mathrm{parent}(f, v) = \begin{cases} \text{the parent of } v \text{ in } f & \text{if } v \text{ is not the root of } f, \\ nil & \text{otherwise.} \end{cases}$
- $\mathrm{anc}(f, v) = \{v' \mid v' \text{ is an ancestor of } v \text{ in } f\}$
- $\mathrm{desc}(f, v, l) = \{v' \mid \lambda(v') = l,\ v' \text{ is a descendant of } v \text{ in } f\}$
- $\mathrm{attr}(v, name) =$ the value of *name* attribute of v

Let c be a connecting node in f. A sequence of nodes from c to the root node r of f, called *connecting path* of c, is defined as $cp(f, c) = [\mathrm{parent}(f, c), \mathrm{parent}(f, \mathrm{parent}(f, c)), \cdots, r]$. For example, consider $f_{e,1}$ shown in Fig. 3.1. Then $cp(f_{e,1}, c_1) = [v_4, v_1]$.

A connecting node c in f has a url attribute, whose value is the URL of the child site connected by c. For example, consider node c_1 in Fig. 3.1. We have $\mathrm{attr}(c_1, \text{"url"}) =$ "http://server2.com/fe2" (see Fig. 3.2). By $CP(f)$, we mean the set of pairs of such a URL and connecting path of c in f, that is,

$$CP(f) = \{\langle \mathrm{attr}(c, \text{"url"}), cp(f, c)\rangle \mid c \text{ is a connecting node in } f\}$$

For example, for the fragment $f_{e,1}$ in Fig. 3.1, $CP(f_{e,1}) = \{\langle \text{http://server2.com/fe2}, [v_4, v_1]\rangle\}$. Let $H = CP(f)$. A function cp_H and a set of URLs $Url(H)$ are defined as follows.

$$cp_H(url) = \begin{cases} cp & \text{if } \langle url, cp\rangle \in H, \\ nil & \text{otherwise.} \end{cases}$$
$$Url(H) = \{url \mid cp_H(url) \neq nil\}$$

Let t be a tree and $f \in F_t$ be a fragment. A sequence of nodes from the root node r of f to the root node r' of t, called *root path* of f, is defined as $rp(t, f) = [\text{parent}(t, r), \text{parent}(t, \text{parent}(t, r)), \cdots, r']$ (if f is the root fragment, $rp(t, f) = nil$). For example, $rp(t_e, f_{e,2}) = [v_4, v_1]$ (Fig. 3.1). Let *url* be the URL of the site storing f, and f' be the parent fragment of f. By the definition, $rp(t, f)$ is obtained by appending the root path of f' to $cp_{CP(f')}(url)$. The i-th node of root path rp is denoted rp_i and the next node of rp_i is denoted $\text{next}(rp_i) = rp_{i+1}$. For example, if $rp = [v_4, v_1]$, then $next(v_4) = rp_2 = v_1$.

Let us now present Master-XSLT. This procedure first sends a tree transducer to each slave site (line 1), then sends the root path to each site, which is used for path precomputation (lines 2 to 5). $S(url)$ in line 4 denotes the site whose url is *url*. Then transforms the root fragment by procedure Transform (line 8, shown later). The last parameter M of Transform is the cache storing the result of predicate evaluations (details are shown in the next section). Finally, all the transformed fragments are merged into one tree, which is the final result (line 10).

Master-XSLT
Input : Tree transducer $Tr = (Q, \Sigma, q_0, R)$, the root fragment f of tree t
Output : Tree $t' = Tr(t)$

1. Send tree transducer Tr to each slave site.
2. $H \leftarrow CP(f)$;
3. **for** each $url \in Url(H)$ **do**
4. Send $cp_H(url)$ to $S(url)$.
5. **end**
6. $v \leftarrow$ the root node of f;
7. $M \leftarrow \emptyset$;
8. $f' \leftarrow$ Transform(Tr, f, v, q_0, nil, M);
9. Wait until a transformed fragment is received from each slave site. Let $f_1, \cdots, f_k$ be the received fragments.
10. Merge f' and $f_1, \cdots, f_k$ into t'.
11. **Return** t';

Next, we present procedure Transform used in line 8 of Master-XSLT (and Slave-XSLT shown later). Let $Tr = (Q, \Sigma, q_0, R)$ be a tree transducer, t be a tree, $f \in F_t$ be a fragment, and rp be the root path of f. To transform the subtree rooted at a node v of f or rp in state q, we need to determine a rule applied to v and obtain the hedge that is the right-hand side of the rule. Such a hedge is denoted by $h(f, Tr, q, v, rp, M)$. More specifically, if there is a pattern *pat* such that $(q, pat) \to h \in R$ and that there is an ancestor v' of v such that v is reachable from v' via *pat* in t, then $h(f, Tr, q, v, rp, M)$ coincides with h, that is,

$$h(f, Tr, q, v, rp, M) = \begin{cases} h & \text{if } \exists pat((q, pat) \to h \in R \wedge \text{Eval-}M_{pat}(f, v, pat, rp, M)), \\ \epsilon & \text{otherwise,} \end{cases} \tag{3.1}$$

where Eval-$M_{pat}(f, v, pat, rp, M)$ is the distributed version of $M_{pat}(T, v, pat)$ and takes the following value (defined in the next section).

$$\text{Eval-}M_{pat}(f, v, pat, rp, M) = \begin{cases} true & \text{if } M_{pat}(t, v, pat) \neq \emptyset, \\ false & \text{otherwise.} \end{cases}$$

The correctness of the above equation is shown in Sect. 3.3. rp and cache M do not appear in the right-hand side, since they are required for only the evaluation of Eval-M_{pat} in a distributed context. As an example of (1), consider the fragment $f_{e,1}$ shown in Fig. 3.1 and the tree transducer Tr of Example 1. Since $(p, \downarrow^*\text{::branch}) \rightarrow x(p\ q) \in R$ and v_1 satisfies pattern "$\downarrow^*$::branch", we have $h(f_{e,1}, Tr, p, v_1, nil, M) = x(p\ q)$. Since $f_{e,1}$ is the root fragment, the fifth parameter is *nil*.

Let us show procedure Transform, which transforms a given fragment recursively according to the definition of tree transducer. In line 2, $St(h)$ denotes the set of states in h. For example, if $h = x(p\ q)$, then $St(h) = \{p, q\}$. Lines 3 to 7 apply rule R1 and lines 8 to 14 apply rule R2 of the definition of tree transducer.

Procedure Transform
Input : Tree transducer $Tr = (Q, \Sigma, q_0, R)$, fragment f, context node v, state q, the root path rp of f, cache M
Output : Hedge h obtained by transforming f with Tr

1. $h \leftarrow h(f, Tr, q, v, rp, M)$;
2. $Q' \leftarrow St(h)$;
3. **if** $Q' \neq \emptyset$ **and** child$(f, v) = \emptyset$ **then**
4. **for** each $q' \in Q'$ **do**
5. Replace node q' in h with ϵ.
6. **end**
7. **end**
8. Let child$(f, v) = \{v'_1, \cdots, v'_k\}$.
9. **for** each $q' \in Q'$ **do**
10. **for** each $v'_i \in$ child(f, v) **do**
11. $h_{v'_i} \leftarrow$ Transform(Tr, f, v'_i, q', rp, M);
12. **end**
13. Replace node q' in h with hedge $h_{v'_1} \cdots h_{v'_k}$.
14. **end**
15. **Return** h;

Finally, we present Slave-XSLT. This procedure runs in each slave site S and transforms the fragment f stored in S. First, we show a definition. Let f be a fragment and $Tr = (Q, \Sigma, q, R)$ be a tree transducer. To transform f, it needs to determine the set of states applied to the root node of f. The set can be obtained by applying rules of Tr to the root path $rp = [rp_1, \cdots, rp_n]$ of f from rp_n to rp_1 (recall that rp_n is the root of the input tree). Formally, let Q' be the initial states applied to rp_n. Then the set of states applied to the root node of f is recursively obtained as follows. First, if $n = 1$, then

$$Q_r(f, Tr, Q', rp, M) = \{q'' \mid q'' \in St(h(f, Tr, q', rp_n, rp, M)), q' \in Q'\}. \tag{3.2}$$

Second, if $n > 1$, then

$$Q_r(f, Tr, Q', rp, M) = \{q'' \mid q'' \in Q_r(f, Tr, Q'', rp', M), \\ Q'' = \{q \mid q \in St(h(f, Tr, q', rp_n, rp, M)), q' \in Q'\}\}, \tag{3.3}$$

where $rp' = [rp_1, \cdots, rp_{n-1}]$ and $h(f, Tr, q', rp_n, rp, M)$ is the hedge defined in (3.1).

Example 2 Let $rp = [v_4, v_1]$ be the root path of $f_{e,2}$ in Fig. 3.1, $Tr = (Q, \Sigma, q_0, R)$ be the tree transducer in Example 1, and $Q' = \{p\}$. Consider computing $Q_r(f_{e,2}, Tr, Q', rp, M)$. By (3), we have

$$Q_r(f_{e,2}, Tr, Q', rp, M) = \{q'' \mid q'' \in Q_r(f_{e,2}, Tr, Q'', [v_4], M)\}, \tag{3.4}$$

where $Q'' = \{q \mid q \in St(h(f_{e,2}, Tr, p, v_1, rp, M))\}$. As shown in Example 1, $(p, \downarrow^*::\text{branch}) \to x(p\,q)$ is applicable to v_1 in state p. Thus we have $h(f_{e,2}, Tr, p, v_1, rp, M) = x(p\,q)$ and $Q'' = \{p, q\}$. Consider the right-hand side of (4). By (2), we have

$$Q_r(f_{e,2}, Tr, Q'', [v_4], M) = \{q'' \mid q'' \in St(h(f_{e,2}, Tr, q', v_4, [v_4], M)), q' \in Q''\}, \tag{3.5}$$

where $Q'' = \{p, q\}$. Since $(p, \downarrow^*::\text{branch}[\downarrow::\text{location}]/\downarrow::\text{clientele}) \to y(p)$ is applicable to v_4 in state p, $h(f_{e,2}, Tr, p, v_4, [v_4], M) = y(p)$. On the other hand, there is no rule applicable to v_4 in state q, thus $h(f_{e,2}, Tr, q, v_4, [v_4], M) = nil$. Thus, we have $Q_r(f_{e,2}, Tr, Q'', [v_4], M) = \{p\}$ by (5), which implies that $Q_r(f_{e,2}, T_r, Q', rp, M) = \{p\}$ by (4). Hence only state p is applicable to the root of $f_{e,2}$.

We now present Slave-XSLT. This procedure first receives tree transducer Tr from the root site and root path rp from the parent site (lines 1 to 2). rp is send from line 4 of Master-XSLT (when the parent site is the root site), or line 6 of Slave-XSLT (when the parent site is a slave site). If f has a child fragment, say f', send the root path of f' to the child site storing f' (lines 4 to 7). From this, each child site can obtain the root path of its own fragment. Then the set of states applied to the root of f is calculated and f is transformed (lines 10 to 13). Finally, the transformed fragment of f is send to the root site (line 14).

Slave-XSLT
Input : Fragment f.
Output : none (transformed fragments are sent to the root site).

1. Wait until tree transducer $Tr = (Q, \Sigma, q_0, R)$ is received from the root site.
2. Wait until the root path rp is received from the parent site.
3. $H \leftarrow CP(f)$;
4. **for** each $url \in Url(H)$ **do**
5. Let rp' be the root path of the fragment in $S(url)$, obtained by appending rp to $cp_H(url)$;
6. Send rp' to $S(url)$.
7. **end**
8. $v \leftarrow$ the root node of f;
9. $M \leftarrow \emptyset$;
10. $Q' \leftarrow Q_r(f, Tr, \{q_0\}, rp, M)$;
11. **for** each $q' \in Q'$ **do**
12. $f'_{q'} \leftarrow$ Transform(Tr, f, v, q', rp, M);

13. **end**
14. Send all $f'_{q'}(q' \in Q')$ to the root site.

3.2 Evaluation of pattern

To check if a node matches a pattern, we have to evaluate M_{pat} and M_{pd} in a distributed environment, as used in (1). Thus we define procedures Eval-M_{pat} and Eval-M_{pd}, which are distributed versions of M_{pat} and M_{pd}, respectively. First, we show several definitions. Let f be a fragment, $rp = [rp_1, \cdots, rp_n]$ be the root path of f, and v be a node of f or rp. The set of parent nodes of v, denoted parent$'(f, v, rp)$, is defined as follows.

$$\text{parent}'(f, v, rp) = \begin{cases} \{rp_1\} & \text{if } v \text{ is the root node in } f \text{ and } rp \neq nil, \\ \{\text{next}(v)\} & \text{if } v \text{ is a node in } rp, \\ \{\text{parent}(f, v)\} & \text{otherwise.} \end{cases}$$

By $Anc(f, v, ax, rp)$ we mean the set of ancestors v' of v such that v is reachable from v' via axis ax, that is,

$$Anc(f, v, ax, rp) = \begin{cases} \text{parent}'(f, v, rp) & \text{if } ax = \downarrow, \\ \text{anc}(f, v) \cup \{rp_1, \cdots, rp_n\} & \text{if } ax = \downarrow^* . \end{cases}$$

For example, let $f_{e,2}$ be the fragment shown in Fig. 3.1, v_7 be the node in $f_{e,2}$, and $rp = [v_4, v_1]$ be the root path of $f_{e,2}$. Then $Anc(f_{e,1}, v_7, \downarrow^*, rp) = \{v_5, v_4, v_1\}$, where v_5 is the ancestor of v_7 in $f_{e,2}$, v_4 and v_1 are the nodes in rp.

Next, we define *cache* used for evaluating predicates. A cache is created in each site (line 9 of Master-XSLT, line 9 of Slave-XSLT). A cache is passed *by reference*, and thus all the procedures running in the same site share the same cache. Let v be a node and pd be a predicate. A string "$v + pd$" is called *predicate inquiry*. This is sent to the site having v to ask if v satisfies pd. Let *query* be a predicate inquiry and $res \in \{true, false\}$ be the result of the predicate inquiry. Then a cache M holds a set of pairs $\langle query, res \rangle$, and $f_M(query)$ denotes the result of *query* to M, that is,

$$f_M(query) = \begin{cases} res & \text{if } \langle query, res \rangle \in M, \\ nil & \text{otherwise.} \end{cases}$$

We present procedure Eval-M_{pat}. Let $pat = ls_1/\cdots/ls_n$ be a pattern, where $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}]$. This procedure decides if a context node v matches *pat* by examining *pat* from ls_n to ls_1 recursively. Lines 1 to 17 check if v satisfies the conditions described in ls_n. If v is a node in rp (i.e., v is in an outside site), for each $j = 1, \cdots, m_n$ predicate inquiry "$v + pd_{n,j}$" is sent to the site having v in case of a cache miss (lines 6 to 9, pat_inq is shown later). If v is a node in f, then this procedure checks if v satisfies $pd_{n,1}, \cdots, pd_{n,m_n}$ by Eval-M_{pd} (line 12, Eval-M_{pd} is shown later). When the checks for ls_n are completed, then the procedure checks if there is an ancestor $v' \in Anc(f, v, ax_n, rp)$ of v such that v' matches $ls_1/\cdots/ls_{n-1}$ (lines 21 to 23).

Procedure Eval-M_{pat}

Input : Fragment f, context node v, pattern $pat = ls_1/\cdots/ls_n$ with $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}]$, the root

path rp of f, cache M

Output : $true$ if v matches pat, $false$ otherwise

1. **if** $\lambda(v) \neq l_n$ **then**
2. **Return** $false$;
3. **end**
4. **for** each $j = 1, \cdots, m_n$ **do**
5. **if** v is a node in rp **then**
6. **if** $f_M(\text{“}v + pd_{n,j}\text{”}) = nil$ **then**
7. Ask the site S having v if v satisfies $pd_{n,j}$, by calling pat_inq(f', v, $pd_{n,j}$, M) in S, where f' is th fragment stored in S. Let res be the result.
8. $M \leftarrow M \cup \langle \text{“}v + pd_{n,j}\text{”}, res \rangle$;
9. **end**
10. $pred\_result \leftarrow f_M(\text{“}v + pd_{n,j}\text{”})$;
11. **else**
12. $pred_result \leftarrow$Eval-$M_{pd}(f, v, pd_{n,j}, M)$;
13. **end**
14. **if** $pred\_result = false$ **then**
15. **Return** $false$;
16. **end**
17. **end**
18. **if** $n = 1$ **then**
19. **Return** $true$;
20. **end**
21. $V \leftarrow Anc(f, v, ax_n, rp)$;
22. $pat' \leftarrow ls_1/\cdots/ls_{n-1}$;
23. **if** Eval-$M_{pat}(f, v', pat', rp, M) = true$ for some $v' \in V$ **then**
24. **Return** $true$;
25. **else**
26. **Return** $false$;
27. **end**

Procedure pat_inq in line 7 is defined as follows. This procedure runs in parallel using native threads.

Procedure pat_inq

Input : Fragment f, context node v, predicate $pred = ls_1/\cdots/ls_n$ with $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}]$, cache M

Output : none

1. $res \leftarrow$Eval-$M_{pd}(f, v, pred, M)$
2. Send res to the calling procedure.

Function h defined by (3.1) uses Eval-M_{pat} to determine the rule (and the hedge) matched by the current node. Although the formal definition of Eval-M_{pat} is a bit complicated due to the distribution of fragments and predicates (lines 4 to 17), we can implement the procedure and the function so that the rule matched by

```
{
    currency : { branch : r1},
    name : {
        client : {
           clientele : r2,
        }
    },
    deals : {
        client : {
           clientele : r3,
        }
    }
}
```

Figure 3.3 Nested hash structure constructed from three patterns

the current node can be identified efficiently, especially if patterns consist of only labels and child axes. For example, suppose that we have the following three rules.

- $r1$: $(q, \downarrow\!::\text{ branch}/\downarrow\!::\text{ currency}) \to h_1$
- $r2$: $(q, \downarrow\!::\text{ clientele}/\downarrow\!::\text{ client}/\downarrow\!::\text{ name}) \to h_2$
- $r3$: $(q, \downarrow\!::\text{ clientele}/\downarrow\!::\text{ client}/\downarrow\!::\text{ deals}) \to h_3$

From the three patterns above, we construct nested hash functions as shown in Fig. 3.3. Here, suppose that the current node v is labeled by "name", its parent v' is labeled by "client", and that the parent v'' of v' is labeled by "clientele". These labels can be obtained immediately by using a root path, even if v' or v'' is not in the fragment having v. Then we can easily identify $r2$ as the rule matched by v, by applying the labels of v, v', v'' to the hash functions in Fig. 3.3. In general, if patterns consist of only labels and child axes, the rule matched by the current node can be identified in $O(|p_{max}|)$, where $|p_{max}|$ is the maximum length (i.e., the number of location steps) of patterns.

Next, we present procedure Eval-M_{pd} used in line 12 of Eval-M_{pat} and line 1 of pat_inq. This procedure evaluates predicates of a pattern. By $Desc(f, v, ax, l)$, we mean the set of descendants v' such that v' is either reachable from v via location step $ax::l$ or a connecting node, that is,

$$Desc(f, v, ax, l) = \begin{cases} \{v' \mid v' \in \text{child}(f, v), \lambda(v') \in \{l, \text{“CONNECT”}\}\} & \text{if } ax = \downarrow, \\ \text{desc}(f, v, l) \cup \text{desc}(f, v, \text{“CONNECT”}) & \text{if } ax = \downarrow^* . \end{cases}$$

For example, consider the fragment $f_{e,1}$ shown in Fig. 3.1. Then $Desc(f_{e,1}, v_1, \downarrow^*, \text{“currency”}) = \{v_2, c_1\}$.

Let $pred = ls_1/\cdots/ls_n$, where $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_i, m_i]$. This procedure decides if a context node v satisfies $pred$ by examining $pred$ from ls_1 to ls_n recursively. First, the procedure calculates the set V of nodes reachable from v via $ax_1::l_1$ (line 1). Then this procedure checks for each $v' \in V$,

a) whether v' satisfies $pd_{1,1}, \cdots, pd_{1,m_1}$, and

b) whether v' satisfies $ls_2/\cdots/ls_n$.

Suppose that v' is a connecting node. Since the checks of (a) and (b) require accessing to outside sites, we use another kind of predicate inquiry of the form "*url* + *pred*" to ask if v' satisfies (a) and (b), where *url* is the value of the url attribute of v' (the result is stored in the same cache M as used in Eval-M_{pat}). Predicate inquiry "*url* + *pred*" is send to $S(url)$ in case of a cache miss (lines 6 to 9, pd_inq is shown later). By this predicate inquiry, f_M("*url* + *pred*") = true if (a) and (b) hold (line 10). If v' is not a connecting node, (a) is obtained in line 12 and (b) is obtained in line 15.

Procedure Eval-M_{pd}
Input : Fragment f, context node v, predicate $pred = ls_1/\cdots/ls_n$ with $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}]$, cache M
Output : *true* if v satisfies *pred*, $false$ otherwise

1. $V \leftarrow Desc(f, v, ax_1, l_1)$;
2. $result \leftarrow false$;
3. **for** each $v' \in V$ **do**
4. **if** $\lambda(v')$ = "CONNECT" **then**
5. $url \leftarrow$ attr(v, "url");
6. **if** f_M("*url* + *pred*") = *nil* **then**
7. Ask $S(url)$ if v' satisfies above (a) and (b), by calling pd_inq($f', pred, M$), where f' is the fragment stored in $S(url)$. Let res be the result.
8. $M \leftarrow M \cup \langle$"*url* + *pred*", $res\rangle$;
9. **end**
10. $result \leftarrow result \vee f_M$("*url* + *pred*");
11. **else**
12. $p\_res \leftarrow true \wedge \bigwedge_{j=1,\cdots,m_1}$ Eval-$M_{pd}(f, v', pd_{1,j}, M)$;
13. **if** $n > 1$ **and** p_res **then**
14. $pred' \leftarrow ls_2/\cdots/ls_n$;
15. $result \leftarrow result \vee$ Eval-$M_{pd}(f, v', pred', M)$;
16. **else if** p_res **then**
17. **Return** *true*;
18. **end**
19. **end**
20. **end**
21. **Return** $result$;

Finally, we present procedure pd_inq used in procedure Eval-M_{pd}. This procedure runs in parallel using native threads.

Procedure pd_inq
Input : Fragment f, predicate $pred = ls_1/\cdots/ls_n$ with $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}]$, cache M
Output : none

1. $v \leftarrow$ the root node of f;

2. **if** $ax_1 = \downarrow$ **then**
3. **if** $l_1 = \lambda(v)$ **then**
4. $p\_res \leftarrow \bigwedge_{j=1,\cdots,m_1}$ Eval-$M_{pd}(f, v, pd_{1,j}, M)$;
5. **if** $n > 1$ **and** p_res **then**
6. $pred' \leftarrow ls_2/\cdots/ls_n$;
7. $res \leftarrow$Eval-$M_{pd}(f, v, pred', M)$;
8. **else if** p_res **then**
9. $res \leftarrow true$;
10. **end**
11. **end**
12. **else**
13. $res \leftarrow$Eval-$M_{pd}(f, v, pred, M)$
14. **end**
15. Send res to the calling site.

Example 3 Let $f_{e,1}$ be the fragment shown in Fig. 3.1, pat=↓*::branch[↓*::name]/↓::currency be a pattern, and v_2 be the node of $f_{e,1}$. Then Eval-$M_{pat}(f_{e,1}, v_2, pat, nil, M)$ is evaluated by the following steps.

1. Check if v_2 matches ↓::currency.
   a. We have $\lambda(v_2)$ = "currency" in line 1 of Eval-M_{pat}.
   b. We have $Anc(f_{e,1}, v_2, \text{"child"}, nil) = \{v_1\}$ in line 21 of Eval-M_{pat}.
   Then Eval-$M_{pat}(f_{e,1}, v_1, \downarrow^*$::branch[↓*::name], $nil, M)$ is called in line 23.
2. By step (1-b) above, check if v_1 matches ↓*::branch[↓*::name].
   a. We have $\lambda(v_1)$ = "branch" in line 1 of Eval-M_{pat}. Since $rp = nil$, Eval-$M_{pd}(f_{e,1}, v_1, \downarrow^*$ ::name, $nil, M)$ is called in line 12.
   b. There is no descendant of v_1 labeled by "name" but there is a connecting node c_1 in $f_{e,1}$. Thus we use a predicate inquiry "*url* + *pred*" in lines 4 to 9 of Eval-M_{pd}, where *url* = "http://server2.com/fe2" and *pred*= "↓*:: name".
   c. If f_M("*url* + *pred*") = *nil* in line 6, $S_{e,2}$ is asked to call pd_inq($f_{e,2}, \downarrow^*$ ::name, M).
   d. In $S_{e,2}$, pd_inq($f_{e,2}, \downarrow^*$::name, M) returns *true* since the label of v_6 is "name".
3. Since the result of (1-b) is true by step (2) above, Eval-$M_{pat}(f_{e,1}, v_2, pat, nil, M)$ returns *true*.

3.3 Correctness of the Method

We show the correctness of our method. We first show the correctness of Eval-M_{pd}. Let t be a tree, f be a fragment of t, v be a node of f, and $pred = ls_1/\cdots/ls_n$ be a predicate with $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pdi, m_i]$. To show the correctness, we have to handle two parameters: the *nesting level* of f and the *size* of *pred*. First, the *nesting level* of f, denoted $ns(f)$, is defined as follows.

$$ns(f) = \begin{cases} 0 & \text{if } f \text{ has no child fragment,} \\ \max\limits_{f'} ns(f') + 1 & \text{otherwise,} \end{cases}$$

where f' ranges over the child fragments of f. Second, the *size* of *pred*, denoted $size(pred)$, is defined as follows.

$$size(pred) = \begin{cases} 1 + \sum_{1 \le j \le m_1} size(pd_{1,j}) & \text{if } pred = ax_1 :: l_1[pd_{1,1}] \cdots [pd_{1,m_1}], \\ \sum_{1 \le i \le n} size(ls_i) & \text{otherwise.} \end{cases}$$

In Lemma 2, we show that Eval-$M_{pd}(f, v, pred, M)$ returns *true* iff $M_{pd}(t, v, pred) \neq \emptyset$ by double induction on $ns(f)$ and $size(pred)$. The following Lemma 1 is required to show the basis cases of Lemma 2.

Lemma 1 Let $ls = ax :: l$ be a location step. Then Eval-$M_{pd}(f, v, ls, M)$ returns *true* iff $M_{pd}(t, v, ls) \neq \emptyset$.

Proof(sketch): We show that Eval-$M_{pd}(f, v, ls, M)$ returns true iff $M_{pd}(t, v, ls) \neq \emptyset$ by induction on $ns(f)$.

Basis : $ns(f) = 0$. Since f has no child fragment, the descendants of v in f coincide with those of v in t. Therefore, Eval-$M_{pd}(f, v, pred, M)$ returns *true* iff $M_{pd}(t, v, pred) \neq \emptyset$.

Induction : Assume as the induction hypothesis that if $ns(f) \le k$, then Eval-$M_{pd}(f, v, ls, M)$ returns *true* iff $M_{pd}(t, v, ls) \neq \emptyset$. Consider the case where $ns(f) = k + 1$. Let S be the site storing f, c be a connecting node of f, S' be the child site of S connected by c, and f' be the fragment stored in S'. Consider the set V of nodes obtained in line 1 of Eval-M_{pd}. If V contains connecting node c, ls is sent to S' by calling pd_inq (line 7 of Eval-M_{pd}). In S', ls is evaluated to the root node r of f' by pd_inq. Since $ns(f') \le k$ and thus Eval-M_{pd} is correct by the induction hypothesis, pd_inq sends *true* to S iff $M_{pd}(t, r, ls) \neq \emptyset$. Hence Eval-$M_{pd}(f, v, ls, M)$ returns *true* iff $M_{pd}(t, v, ls) \neq \emptyset$. □

We next have the following lemma, which shows the correctness of Eval-M_{pd}.

Lemma 2 Let $pred = ls_1/\cdots/ls_n$, where $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}]$ $(1 \le i \le n)$. Then Eval-$M_{pd}(f, v, pred, M)$ returns *true* iff $M_{pd}(t, v, pred) \neq \emptyset$.

Proof(sketch): We show that Eval-$M_{pd}(f, v, pred, M)$ returns true iff $M_{pd}(t, v, pred) \neq \emptyset$ by double induction on $size(pred)$ and $ns(f)$.

Basis : First, consider the case where $ns(f) = 0$ and $size(pred) = j$. Since f has no child fragment, the descendants of v in f coincide with those of v in t. Therefore, Eval-$M_{pd}(f, v, pred, M)$ returns *true* iff $M_{pd}(t, v, pred) \neq \emptyset$. Consider next the case where $ns(f) = k$ and $size(pred) = 1$. Since $size(pred) = 1$, by Lemma 1 Eval-$M_{pd}(f, v, pred, M)$ returns *true* iff $M_{pd}(t, v, pred) \neq \emptyset$.

Induction : Assume the following as the induction hypotheses.

2.1. If $ns(f) = x + 1$ and $size(pred) \le y$, then Eval-$M_{pd}(f, v, pred, M)$ returns *true* iff $M_{pd}(t, v, pred) \neq \emptyset$.
2.2. If $ns(f) \le x$ and $size(pred) = y + 1$, then Eval-$M_{pd}(f, v, pred, M)$ returns *true* iff $M_{pd}(t, v, pred) \neq \emptyset$.

Consider the case where $ns(f) = x + 1$ and $size(pred) = y + 1$. Let $pred = ls_1/\cdots/ls_n$, where $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}]$. Moreover, let S be the site storing f, c be a connecting node of f, S' be the child site of S connected by c, and f' be the fragment stored in S'. Consider the set V of nodes obtained in line 1 of Eval-M_{pd}.

- If V contains connecting node c, *pred* is sent to S' by calling pd_inq (line 7 of Eval-M_{pd}).
- The other (non-connecting) nodes in V are (a) evaluated with each predicate $pd_{1,1}, \cdots, pd_{1,m_1}$ (line 12 of Eval-M_{pd}) and (b) evaluated with $pred' = ls_2/\cdots/ls_n$ (line 15 of Eval-M_{pd}).

In S', *pred* is evaluated to the root node r of f' by pd_inq. Since $ns(f') \leq x$ and thus Eval-M_{pd} is correct by the induction hypothesis 2.2, pd_inq sends *true* to S iff $M_{pd}(t, r, pred) \neq \emptyset$. As for non-connecting nodes, since $size(pd_{1,j}) \leq y$ for each $j = 1, \cdots, m_1$, (a) is correct by induction hypothesis 2.1. Since $size(pred') \leq y$, (b) is correct similarly. Thus, the evaluation is correct for each $v' \in V$. Hence Eval-$M_{pd}(f, v, pred, M)$ returns *true* iff $M_{pd}(t, v, pred) \neq \emptyset$. □

Let $pat = ls_1/\cdots/ls_n$ be a pattern with $ls_i = ax_i :: l_i[pd_{i,1}]\cdots[pd_{i,m_i}]$ and rp be the root path of f. We now have the following theorem.

Theorem 1 Eval-$M_{pat}(f, v, pat, rp, M)$ returns *true* iff $M_{pt}(t, v, pat) \neq \emptyset$.

Proof(sketch): Since the ancestors of v in t coincide with the union of (a) the ancestors of v in f and (b) the nodes of rp, Eval-$M_{pat}(f, v, sel(pat), rp, M)$ returns *true* iff $M_{pat}(t, v, sel(pat)) \neq \emptyset$. By Lemma 2, for each $j = 1, \cdots, m_n$, Eval-$M_{pd}(f, v, pd_{n,j}, M)$ (line 12 of Eval-M_{pat}) returns *true* iff $M_{pd}(t, v, pd_{n,j}) \neq \emptyset$. Similarly, pat_inq$(f', v, pd_{n,j}, M)$ (line 7 of Eval-M_{pat}) returns *true* iff $M_{pd}(t, v, pd_{n,j}) \neq \emptyset$. Hence Eval-$M_{pat}(f, v, pat, rp, M)$ returns *true* iff $M_{pt}(t, v, pat) \neq \emptyset$. □

Table 3.1 Summary of XSLT patterns

		W3C Recommendation (XSLT 1.0)	W3C Recommendation (XSLT 2.0)	XSLT 2nd Ed.
Pattern using no predicate	Single label	35	41	127
	Pattern using labels and child axis	7	6	5
	Pattern using "//"	2	2	0
Pattern using predicates		0	1	0
Other (@text, etc.)		7	7	12
Total		51	57	144

3.4 Comparison with Distributed XPath Evaluation Algorithms

Let us consider the difference between distributed XPath evaluation algorithms and our method, from the perspective of XSLT pattern evaluation. First of all, we have the following observations about XPath and XSLT evaluations.

(a) Distributed XPath evaluation algorithms are designed for processing a *single* XPath query efficiently, while an XSLT stylesheet usually contains more than one pattern.

(b) In general, a pattern used in an XSLT stylesheet is much simpler than an XPath query. Table 3.1 shows a summary of the patterns appearing in XSLT 1.0/2.0 W3C recommendations[*1] and the examples used

[*1] http://www.w3.org/TR/xslt/ (XSLT 1.0) and http://www.w3.org/TR/xslt20/ (XSLT 2.0)

in [16]. This table indicates that (i) most patterns use only labels and child axes and that (ii) few patterns use "//" or predicates.

Assuming (a) and (b), the main difference between distributed XPath evaluation algorithms and our method is that, w.r.t. XSLT pattern evaluation, the computation cost of the former algorithms grow proportional to the number of patterns in an XSLT stylesheet while the latter is not. In the following, we compare the algorithm in [4, 5, 6] with our method, since no formal computation cost is presented in [8, 10].

Given a tree t and an XPath query q, the XPath evaluation algorithm [4, 5, 6] traverses t and computes, for each node v in t, several vectors of size $O(|q|)$ which hold the evaluation values of subexpressions of q at v. Thus, to evaluate an XSLT stylesheet xs on t by using the XPath evaluation algorithm, we have to do the following.

1. For each XSLT pattern *pat* in xs, calculate vectors of each node for *pat* by using the algorithm.
2. Perform a top-down XSLT transformation along with xs. During the transformation, the template matched by current node v can be identified by the vectors associated with v.

Now let us consider the parallel computation costs of evaluating XSLT patterns by the above approach and our method in detail. Let k be the number of distinct patterns in xs, *pat* be the longest pattern of the patterns in xs, and f_{max} be the maximum fragment of t. First, consider the parallel computation cost of the above approach. Since the parallel computation cost per pattern is in $O(|pat| \cdot |f_{max}|)$ [6], the parallel computation cost of evaluating k patterns is in $O(k \cdot |pat| \cdot |f_{max}|)$[*2]. Second, consider the parallel computation cost of evaluating XSLT patterns by our method. To evaluate XSLT patterns by our method, we need a root path for each fragment. Assuming that the height of t is in $O(\log |t|)$, the computation cost of obtaining the root path of a fragment is in $O(\log |t|)$. Then, given the root path of a fragment, consider the parallel computation cost of evaluating XSLT patterns. This cost depends on the size of transformed fragment, but for simplicity we assume that the size of the transformed result of f_{max} is in $O(|f_{max}|)$ (this is not too restrictive since the resulting tree transformed by XSLT is usually smaller then the input tree). Moreover, since an XSLT pattern is simple as mentioned in (b), we can assume that the template matched by the current node can be identified in $O(|pat|)$ as shown in Sect. 3.2. Hence the parallel computation cost of evaluating the patterns in xs by our method is in $O(\log |t| + |pat| \cdot |f_{max}|)$.

Thus, the parallel computation cost of evaluating XSLT patterns by the above approach grows proportionally to the number of distinct patterns in an XSLT stylesheet, which is undesirable since an XSLT stylesheet may contain arbitrary number of templates and patterns. On the other hand, our method have to pay the cost of preparing a root path, but this is usually small in practice, e.g., the height of the DBLP XML data[*3] (1.52GB) is only 7 and that of XMark[*4] is 13 regardless the size of data. Therefore, we believe that our method is more suitable for performing XSLT transformations unless an XML tree has an extremely deep structure.

[*2] This complexity remains the same if *pat* uses no predicate and consists of only labels and child axes.

[*3] http://dblp.uni-trier.de/xml/dblp.xml.gz

[*4] http://www.xml-benchmark.org/

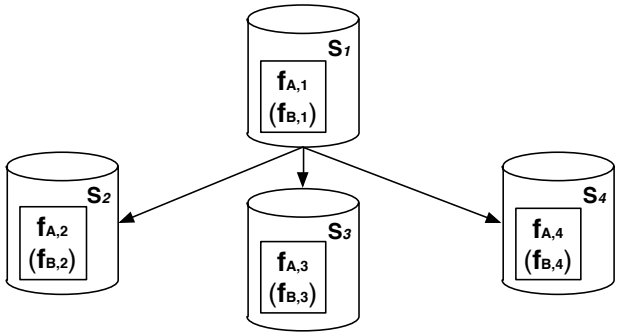

Figure 4.1 Four sites storing the fragments in evaluation experiment

Chapter 4

Evaluation Experiment

In this chapter, we present experimental results on our method. We implemented our method in Ruby 1.9.3. We used 4 Linux machines (S_1 to S_4), distributed over a local LAN (100base-TX). Each machine has a 2.4GHz Intel Xeon CPU and 4GB of memory. First, we generated five XML documents of different sizes by XMark[13]. Since all the sites transform their fragments in parallel, the efficiency of our method may depend on whether the sizes of fragments are even or not. Thus, we create two datasets (A) and (B) from the XMark documents. In dataset (A), the sizes of the fragments are relatively even (Table. 4.1), while in dataset (B) the root fragment is remarkably heavy (Table. 4.2). Tables 4.3 and 4.4 show the root nodes of the fragments, where $f_{A,1}$ ($f_{B,1}$) is the root fragment and has three child fragments $f_{A,2}$, $f_{A,3}$ and $f_{A,4}$ (resp., $f_{B,2}$, $f_{B,3}$ and $f_{B,4}$). The four sites S_1 to S_4 are configured as shown in Fig. 4.1, where S_1 is the root site storing $f_{A,1}$ ($f_{B,1}$) and S_i (i=2,3,4) is a slave site storing $f_{A,i}$ ($f_{B.i}$).

We used eleven synthetic XSLT stylesheets denoted $s_0, s_{10}, \cdots, s_{100}$, generated by our Ruby program. Each s_i has the following properties ($0 \leq i \leq 100$).

1. For every element m of an XML document, s_i has at least one template applicable to m.
2. s_i consists of 218 templates and i% of the templates have a pattern having one predicate (the rest of the templates have a pattern having no predicate).
3. The average length of each selection path is 6 and the average length of each predicate is 5.

Table 4.1 Sizes of distributed XML documents (dataset A)

	Fragment size				
f option	$f_{A,1}$ (root)	$f_{A,2}$	$f_{A,3}$	$f_{A,4}$	Total size
0.5	6.0MB	27.3MB	13.9MB	7.9MB	55.3MB
1.0	12.2MB	54.8MB	27.8MB	16.1MB	111.0MB
1.5	18.6MB	82.2MB	42.2MB	23.9MB	167.0MB
2.0	24.8MB	109.7MB	56.1MB	32.0MB	222.8MB
2.5	31.0MB	137.4MB	70.1MB	40.6MB	279.1MB

4. The predicates of each template are distinct.

We measure the response times of a centralized method and our transformation method. In the centralized method, three child fragments are first sent to the root site S_1, then root and child fragments are merged into one document t and an XSLT transformation is performed on t in S_1. We have the following two settings of evaluation experiments.

a) Fix the stylesheet and measure the response time under various sizes of XML documents.
b) Fix the XML document and measure the response time of different stylesheets.

We used the stylesheet s_{10} under setting (a). The results are shown in Figs. 4.2 and 4.3. For dataset (A), our method is about 6 times faster than the centralized method. Even for dataset (B), our method is about 1.8 times faster than the centralized method. Under setting (b), we used the distributed XML whose total size is 55.3MB. Figures 4.4 and 4.5 show the results. Our method is faster than centralized method regardless the stylesheets. These suggest that our method works well for distributed XML documents.

Tables 4.5 and 4.6 show the details of the response time of the centralized method under setting (a). The tables show the following.

- The details of the response time of the centralized method
  - Time for transferring fragments to the root site
  - Time for merging the fragments into one XML document
  - Time for transforming the merged document
- The response time of our method

These tables show that the response time of our method is smaller than the transformation time of the centralized method for any cases. This suggests that our method is applicable to non-distributed XML documents, by partitioning such documents into fragments and transform them in parallel.

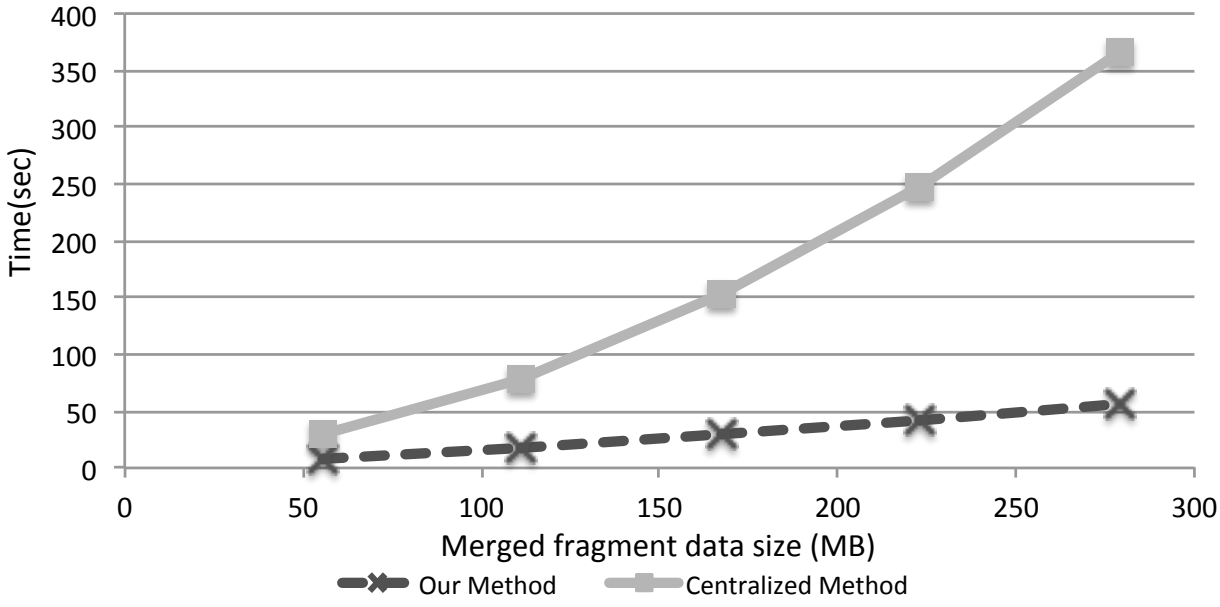

Figure 4.2 Experimental result of (a) (dataset A)

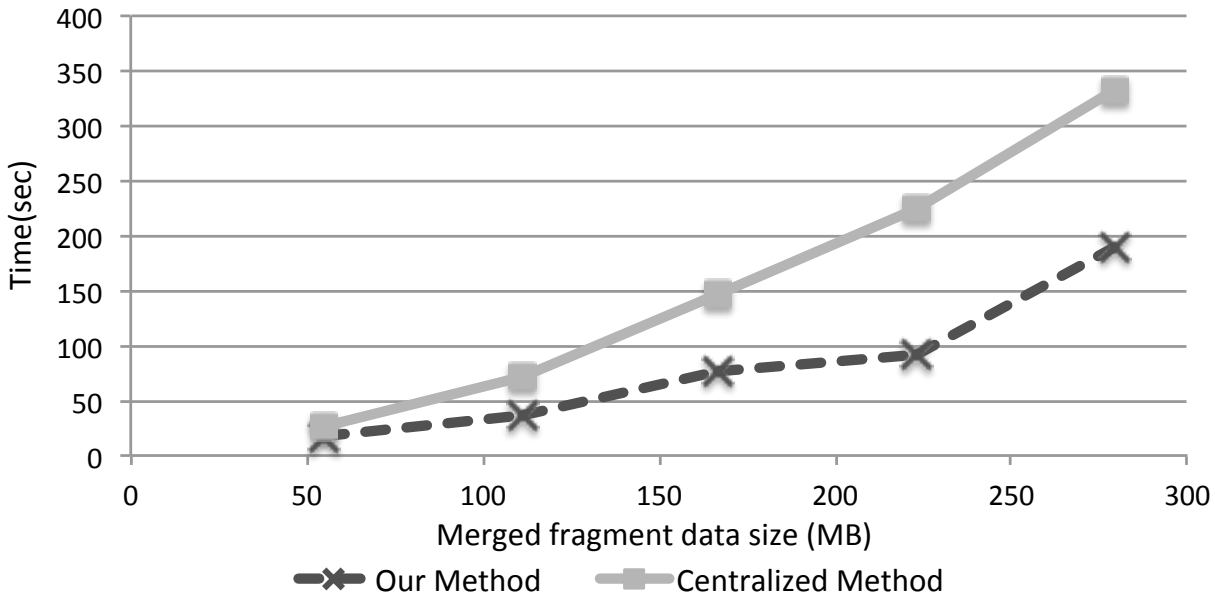

Figure 4.3 Experimental result of (a) (dataset B)

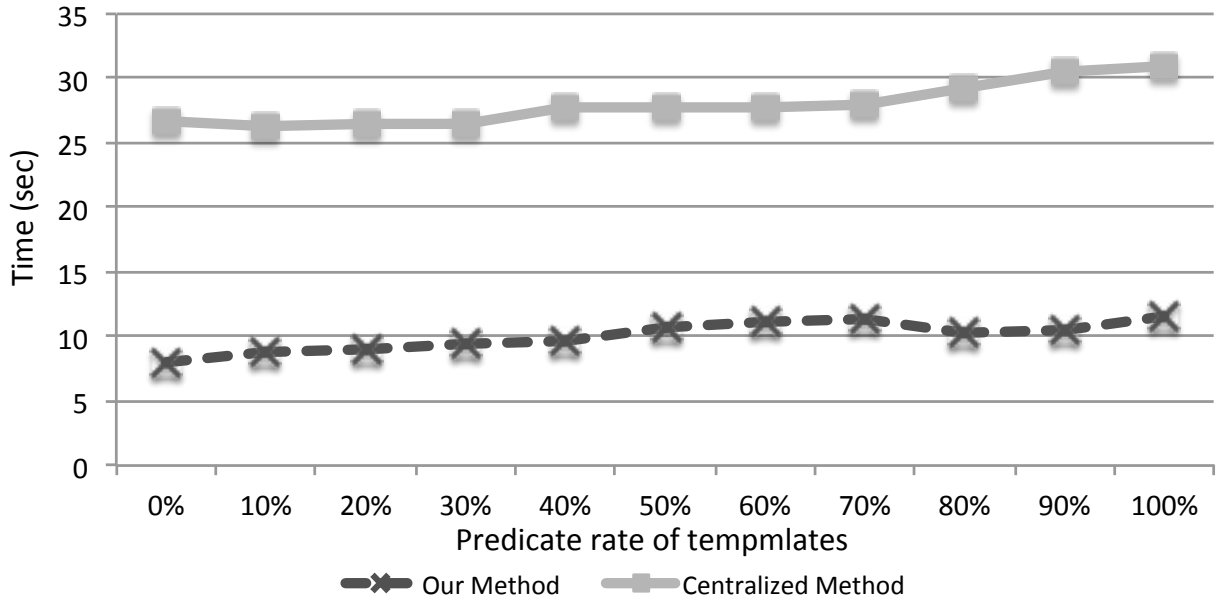

Figure 4.4 Experimental result of (b) (dataset A)

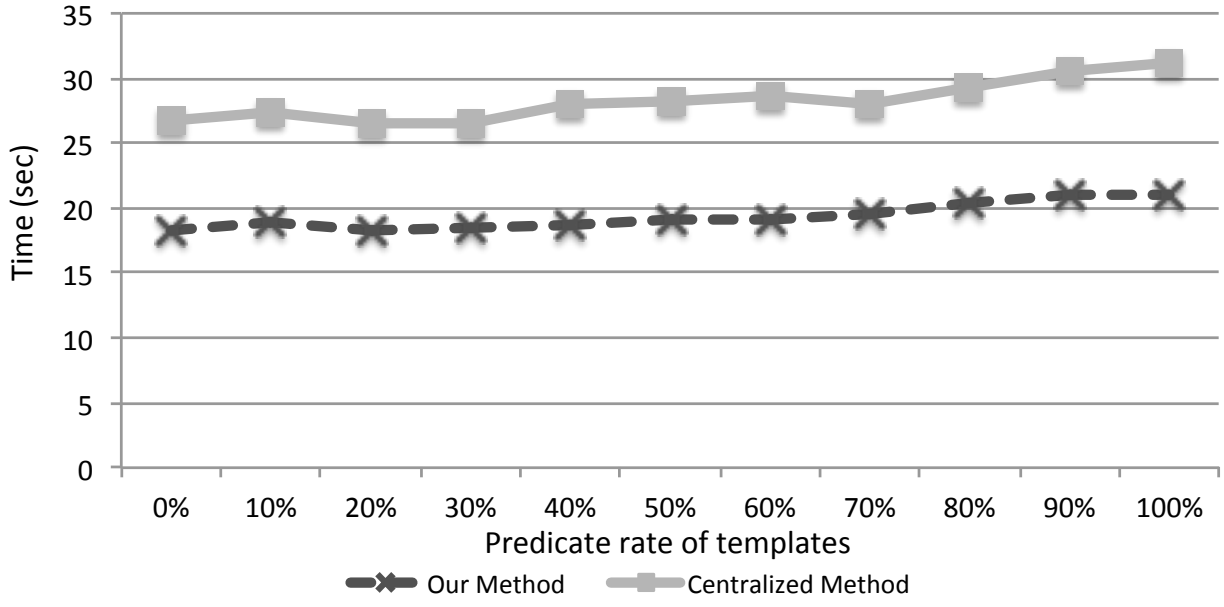

Figure 4.5 Experimental result of (b) (dataset B)

Table 4.2 Sizes of distributed XML documents (dataset B)

	Fragment size				
f option	$f_{B,1}$ (root)	$f_{B,2}$	$f_{B,3}$	$f_{B,4}$	Total size
0.5	44.5MB	2.5MB	2.7MB	5.4MB	55.3MB
1.0	89.6MB	5.0MB	5.6MB	10.8MB	111.0MB
1.5	134.7MB	7.5MB	8.4MB	16.3MB	167.0MB
2.0	179.8MB	10.1MB	10.9MB	21.8MB	222.8MB
2.5	225.4MB	12.6MB	13.8MB	27.3MB	279.1MB

Table 4.3 The root nodes of the fragments (dataset A)

Fragments	Label of nodes
$f_{A,1}$	site
$f_{A,2}$	regions
$f_{A,3}$	open_auctions
$f_{A,4}$	closed_auctions

Table 4.4 The root nodes of the fragments (dataset B)

Fragments	Label of nodes
$f_{B,1}$	site
$f_{B,2}$	asia
$f_{B,3}$	australia
$f_{B,4}$	people

Table 4.5 Details of response time for dataset A (sec)

	Centralized method				Our method
f option (datasize)	Transfer	Merge	Transform	Total	
0.5 (55.3MB)	1.88	0.73	27.04	29.65	7.99
1.0 (111.0MB)	4.71	1.45	73.04	79.2	17.94
1.5 (167.0MB)	8.63	2.51	142.87	154.01	28.98
2.0 (222.8MB)	13.85	2.89	230.06	246.8	41.53
2.5 (279.1MB)	20.05	3.62	343.42	367.09	56.02

Table 4.6 Details of response time for dataset B (sec)

	Centralized method				Our method
f option (datasize)	Transfer	Merge	Transform	Total	
0.5 (55.3MB)	0.67	0.79	24.85	26.31	17.69
1.0 (111.0MB)	1.69	1.85	68.55	72.09	37.13
1.5 (167.0MB)	3.02	2.44	141.26	146.72	77.05
2.0 (222.8MB)	4.91	3.13	215.88	223.92	91.4
2.5 (279.1MB)	6.8	4.05	321.74	332.59	189.01

Chapter 5

Conclusion

In this thesis, we proposed a method for performing XSLT transformation for distributed XML documents. The experimental results suggest that our method work well for distributed XML documents.

However, we have a lot of future work to do. First, in this thesis the expressive power of XSLT is restricted to extended unranked top-down tree transducer. In particular, we have to handle XSLT instructions/functions of Type B in Table 1.1 carefully in order to extend the expressive power of our method. Another future work relates to experimentation. In our experimentation we use only three synthetic XSLT stylesheets. Thus we need to make more experiments using real-world XSLT stylesheets.

Acknowledgements

The author would first like to express my sincere gratitude to warm encouragement and support of my adviser, Associate Professor Nobutaka Suzuki in pursuing this study. Completion of this thesis would be impossible without his help. The author would especially like to thank Assistant Professor Kei Wakabayashi for his valuable suggestions and support. The author is also grateful to Associate Professor Tetsuo Sakaguchi for his daily perceptive comments.

Bibliography

[1] Abiteboul, S., Gottlob, G. and Manna, M.: Distributed XML design, *JCSS*, Vol. 77, No. 6, pp. 936–964 (2011).

[2] Abiteboul, S., Gottlob, G. and Manna, M.: Distributed XML design, *Proc. PODS*, pp. 247–258 (2009).

[3] Bremer, J. M. and Gertz, M.: On distributing XML repositories, *Proc. WebDB*, pp. 73–78 (2003).

[4] Buneman, P., Cong, G., Fan, W. and Kementsietsidis, A.: Using Partial Evaluation in Distributed Query Evaluation, *Proc. VLDB*, VLDB Endowment, pp. 211–222 (2006).

[5] Cong, G., Fan, W. and Anastasios: Distributed Query Evaluation with Performance Guarantees, *Proc. SIGMOD*, pp. 509–520 (2007).

[6] Cong, G., Fan, W., Kementsietsidis, A., Li, J. and Liu, X.: Partial Evaluation for Distributed XPath Query Processing and Beyond, *TODS*, Vol. 37, No. 4, pp. 32:1–32:43 (2012).

[7] Kepser, S.: A simple proof for the Turing-completeness of XSLT and XQuery, *Extreme Markup Languages* (2004).

[8] Kido, K., Amagasa, T. and Kitagawa, H.: Processing XPath Queries in PC-Clusters Using XML Data Partitioning, *Proceedings of the 22nd International Conference on Data Engineering Workshops*, pp. 11–16 (2006).

[9] Kling, P., Özsu, M. T. and Daudjee, K.: Generating efficient execution plans for vertically partitioned XML databases, *PVLDB*, Vol. 4, No. 1, pp. 1–11 (2010).

[10] Kurita, H., Hatano, K., Miyazaki, J. and Uemura, S.: Efficient Query Processing for Large XML Data in Distributed Environments, *AINA'07*, pp. 317–322 (2007).

[11] Martens, W. and Neven, F.: Typechecking Top-Down Uniform Unranked Tree Transducers, *Proc. ICDT*, Lecture Notes in Computer Science, Vol. 2572, Springer Berlin Heidelberg, pp. 64–78 (2003).

[12] Ogden, P., Thomas, D. and Pietzuch, P.: Scalable XML Query Processing Using Parallel Pushdown Transducers, *PVLDB*, Vol. 6, No. 14, pp. 1738–1749 (2013).

[13] Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I. and Busse, R.: XMark: a benchmark for XML data management, *Proc. VLDB*, pp. 974–985 (2002).

[14] Stefanescu, D. C., Thomo, A. and Thomo, L.: Distributed evaluation of generalized path queries, *Proc. SAC*, pp. 610–616 (2005).

[15] Suciu, D.: Distributed query evaluation on semistructured data, *TODS*, Vol. 27, No. 1, pp. 1–62 (2002).

[16] Tidwell, D.: *XSLT, 2nd Edition*, O'Reilly Media (2008).

[17] Zavoral, F. and Dvorakovam, J.: Perfomance of XSLT processors on large data sets, *Proc. ICADIWT*, pp. 110–115 (2009).