

# An XSLT Transformation Method for Distributed XML\*

HIROKI MIZUMOTO<sup>1,a)</sup> NOBUTAKA SUZUKI<sup>1,b)</sup>

Received: September 20, 2014, Accepted: January 9, 2015

**Abstract:** Recently, the sizes of XML documents have rapidly been increasing. *Distributed XML* is a novel form of XML document, in which an XML document is partitioned into fragments and managed separately in plural sites. Distributed XML documents can often be managed more easily than a single large document, according to geographical and/or administrative factors. In this paper, we consider performing XSLT transformation efficiently for distributed XML. To obtain an efficient method for this transformation, we devise an efficient evaluation method for XSLT pattern, which is a subset of XPath, and integrate it into top-down XSLT transformation. To evaluate an XSLT pattern for node  $v$  in a distributed environment, the site having  $v$  may need to access other sites many times. We propose two novel techniques to reduce such accesses; (1) precomputation of ancestors and (2) cache for predicate evaluation. We implemented our method in Ruby and made evaluation experiments. This result suggests that our method is more efficient than a centralized approach.

**Keywords:** XSLT, distributed XML, tree transducer

## 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* [1], [2], [5], [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** and **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 paper, 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 [20]. 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 order to perform an XSLT transformation in a distributed approach, we have to evaluate XSLT patterns in a distributed en-

vironment. Since XSLT pattern is a subset of XPath, distributed XPath evaluation algorithms (e.g., Refs. [4], [5], [6]) would be applicable to such XSLT pattern evaluations. However, these algorithms focus on processing a single XPath query efficiently, while an XSLT stylesheet may contain a number of patterns. This difference may cause no little overhead w.r.t. XSLT pattern evaluation (this is further discussed in Section 3.4). On the other hand, an XSLT pattern is usually much simpler than an XPath query, and thus an XSLT pattern can be evaluated more efficiently than a usual XPath query. These properties of XSLT stylesheet should be taken into account in order to devise an efficient XSLT transformation method for distributed XML.

Therefore, we adopt another XSLT transformation strategy in which each XSLT pattern is evaluated *on the fly* along with top-down XSLT transformation. To achieve this strategy, we have to cope with XSLT patterns evaluated over more than one fragment. 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 and 2, and suppose that we have the following XSLT template.

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

```

\*The previous versions of this paper are Refs. [13] and [14].

<sup>1</sup> University of Tsukuba, Tsukuba, Ibaraki 305–8550, Japan

<sup>a)</sup> s1321653@u.tsukuba.ac.jp

<sup>b)</sup> nsuzuki@slis.tsukuba.ac.jp

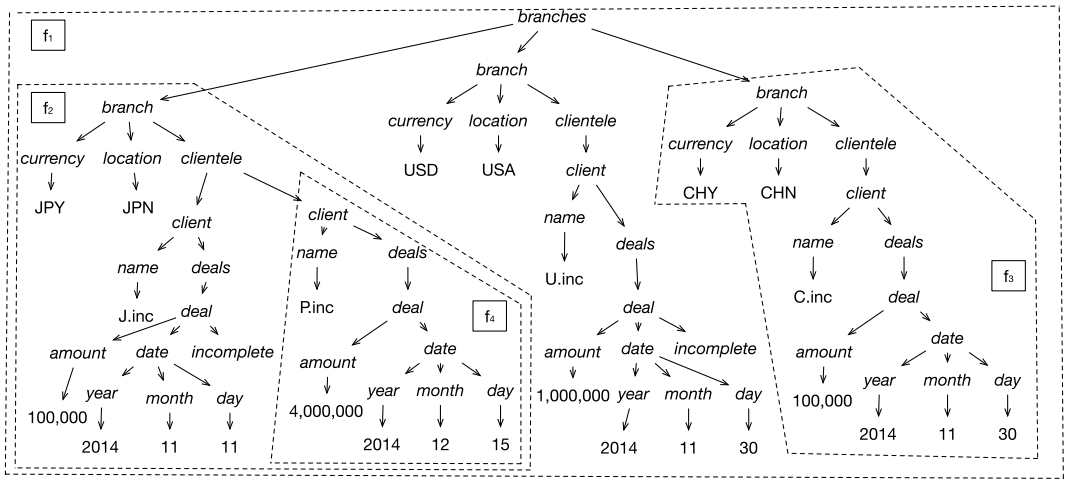


Fig. 1 Multinational corporation clientele.

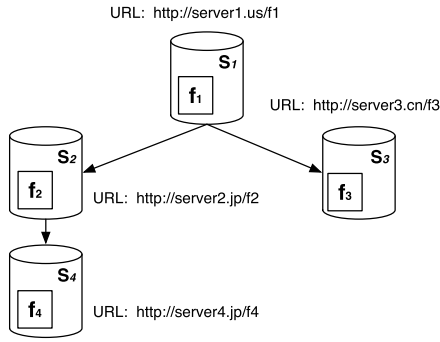


Fig. 2 Four sites storing the fragments in Fig. 1.

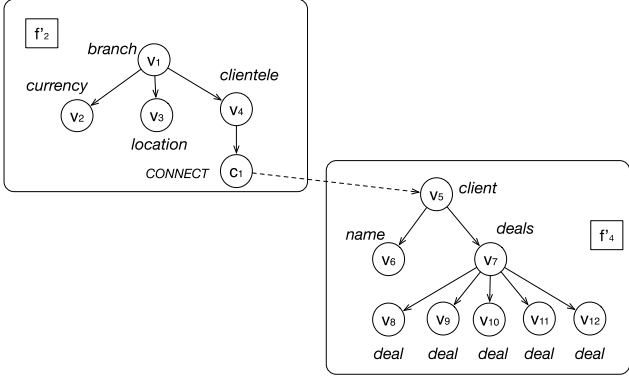


Fig. 3 Sibling nodes labeled as same.

</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  $v_8, \dots, v_{12}$  (Fig. 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. 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. 4, we can easily obtain the list of sales amounts associated with currency of incomplete deals (Fig. 5). Note that such a result cannot be obtained if we use XPath instead of XSLT.

Since XSLT is Turing complete [8], it is hard to plan a complete strategy of XSLT transformation for distributed XML. In this paper, 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 Ref. [12]. 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 paper 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 classifies the instructions/functions of XSLT 1.0 into two types A and B. Here,

```

<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>

```

Fig. 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>

```

Fig. 5 List of sales amounts with currency of incomplete deals.

Table 1 XSLT 1.0 instructions and functions.

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

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 access 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.

### Related Work

A distribution design of XML documents is firstly proposed in Ref. [3]. However, the study makes few discussions on XML query processing. There have been several studies on evaluations of XPath and other languages for distributed XML. References [4], [5], [6], [9], [11] propose efficient XPath evaluation algorithms for distributed XML. Given a tree  $t$  and an XPath query  $q$ , the algorithm in Refs. [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 Ref. [11] partitions an XML tree into fragments, selects appropriate fragments containing answers to the query, then performs a query process-

ing on the fragments in parallel. Reference [9] proposes, assuming that an XML tree is stored in relational tables, a scheme for parallel processing of XML tree using PC-clusters. These XPath evaluation methods may be used for evaluating XSLT patterns, but this is not necessarily a good strategy due to the difference between XPath query and XSLT patterns of XSLT stylesheets (details are discussed in Section 3.4). Reference [10] proposes a method for evaluating XQ, a subset of XPath. The paper assumes that a query is evaluated in a vertically partitioned XML databases, while our method assumes that an XML document is partitioned into subtrees. Reference [17] considers a regular path query evaluation in an distributed environment. Reference [15] 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. Reference [18] extensively studies the complexities of regular path query and structural recursion over distributed semistructured data. Besides query languages, Refs. [1] and [2] 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.

## 2. Definitions

Since our method is based on unranked top-down tree transducer, we first show related definitions. Let  $\Sigma$  be a set of labels. By  $T_\Sigma$  we mean the set of unranked  $\Sigma$ -trees. A tree whose root is labeled with  $a \in \Sigma$  and has  $n$  subtrees  $t_1, \dots, t_n$  is denoted by  $a(t_1 \dots t_n)$ . In the following, we always mean  $\Sigma$ -tree whenever we say tree. A *hedge* is a finite sequence of trees. The set of hedges is denoted by  $H_\Sigma$ . For a set  $Q$ , by  $H_\Sigma(Q)$  we mean the set of  $\Sigma$ -hedges such that leaf nodes can be labeled with elements from  $Q$ . In the following, we use  $t, t_1, t_2, \dots$  to denote trees and  $h, h_1, h_2, \dots$  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  $\Sigma$

Predicate ::= '[' LocationPath '['

From the above definition, a pattern  $pat$  can be expressed as  $pat = ls_1 / \dots / ls_n$ , where  $ls_i = ax_i :: l_i [pd_{i,1}] \dots [pd_{i,m_i}]$ ,  $ax_i$  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,  $sel(pat) = ax_1 :: l_1 / \dots / ax_n :: l_n$ .

Let  $t$  be a tree,  $pat = ls_1 / \dots / ls_n$  be a pattern with  $ls_i = ax_i :: l_i [pd_{i,1}] \dots [pd_{i,m_i}]$  ( $1 \leq i \leq n$ ), 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.

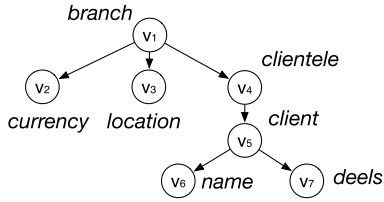


Fig. 6 Tree  $t_e$ .

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

where  $pat' = ls_1 / \dots / ls_{n-1}$  and  $V(t, v, ls_i)$  is the set of ancestors  $v'$  of  $v$  such that  $v$  is reachable from  $v'$  via  $ls_i$ , defined as follows. First, if  $i = 1$  (i.e., the leftmost location step), then

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

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

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

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

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

For example, let  $t_e$  be the tree shown in Fig. 6,  $pat = \downarrow^*::branch/\downarrow::currency$  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 paper, we use an extended version of the unranked tree transducer used in Ref. [12]. Formally, a tree transducer is a quadruple  $(Q, \Sigma, q_0, R)$ , 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 [12],  $pat$  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  $t = a(t_1 \dots t_n)$  and there is a rule  $(q, pat) \rightarrow h$  in  $R$  with  $M_{pat}(t, a, pat) \neq \emptyset$  for some pattern  $pat$ , some  $q \in Q$ , and

```

<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]/clientele"
  mode="p">
  <y>
    <xsl:apply-templates mode="p" />
  </y>
</xsl:template>

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

Fig. 7 An example XSLT script.

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  $Tr^p(t_1) \dots Tr^p(t_n)$ .

- R3: If  $M_{pat}(t, a, pat) = \emptyset$  for every pattern  $pat$ , every  $q \in Q$ , and every  $h \in H_\Sigma, (q, pat) \rightarrow h$  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

$$Q = \{p, q\},$$

$$\Sigma = \{\text{branch, currency, location, clientele, client, name, deals, } x, y, z\},$$

$$R = \{(p, \downarrow^*::branch) \rightarrow x(pq), (q, \downarrow^*::currency) \rightarrow z,$$

$$(p, \downarrow^*::branch[\downarrow::location]/\downarrow::clientele) \rightarrow y(p),$$

$$(p, \downarrow^*::branch/\downarrow^*::client) \rightarrow x(y)\}.$$

$Tr$  corresponds to the XSLT script shown in Fig. 7. For example, consider the rule  $(p, \downarrow^*::branch) \rightarrow x(pq)$  in  $R$ . This corresponds to the first template in Fig. 7. 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. 6 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  $(p, \downarrow^*::branch) \rightarrow x(pq)$  is applied to  $t_e$  and we obtain the tree shown in Fig. 8 (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  $(p, \downarrow^*::branch[\downarrow::location]/\downarrow::clientele) \rightarrow y(p)$  can be applied to  $t_1$ , we obtain the tree shown in Fig. 8 (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. 8 (3).

In this paper, 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 is partitioned into four fragments,  $f_1, f_2, f_3, f_4$ . We allow

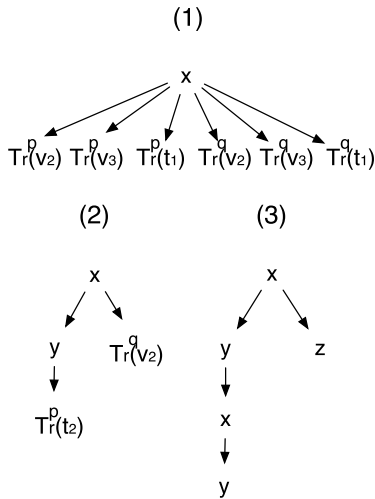


Fig. 8 Tree transformation by  $Tr$ .

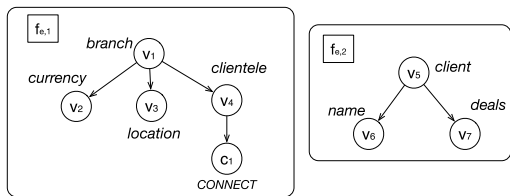


Fig. 9 Fragments  $f_{e,1}, f_{e,2}$  of  $t_e$ .

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. 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. 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. 9 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. 2  $S_1$  has two child sites  $S_2$  and  $S_3$ .

### 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  $pat = ls_1 / \dots / ls_n$  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  $ls_1 / \dots / ls_n$  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.

For example, consider the fragments and the sites shown in Fig. 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.

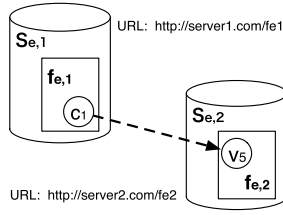
- $child(f, v) = \{v' \mid v' \text{ is a child of } v \text{ in } f\}$
- $parent(f, v) = \begin{cases} \text{the parent of } v \text{ in } f & \text{if } v \text{ is not the root of } f, \\ nil & \text{otherwise.} \end{cases}$
- $anc(f, v) = \{v' \mid v' \text{ is an ancestor of } v \text{ in } f\}$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

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  $cp(f, c) = [parent(f, c), parent(f, parent(f, c)), \dots, r]$ . For example, consider  $f_{e,1}$  shown in Fig. 9. 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. 9. We have  $attr(c_1, “url”) = “http://server2.com/fe2”$  (see Fig. 10). By  $CP(f)$ , we mean the set of pairs of such a URL and connecting path of  $c$  in  $f$ , that is,

$$CP(f) = \{\{attr(c, “url”), cp(f, c)\} \mid c \text{ is a connecting node in } f\}$$

For example, for the fragment  $f_{e,1}$  in Fig. 9,  $CP(f_{e,1}) = \{\langle http://server2.com/fe2, [v_4, v_1] \rangle\}$ . Let  $H = CP(f)$ . A function  $cp_H$  and a set of URLs  $Url(H)$  are defined as follows.


 Fig. 10 Sites  $S_{e,1}$  and  $S_{e,2}$ .

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

$$Url(H) = \{url \mid cp_H(url) \neq nil\}$$

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  $rp(t, f) = [\text{parent}(t, r), \text{parent}(t, \text{parent}(t, r)), \dots, r']$  (if  $f$  is the root fragment,  $rp(t, f) = nil$ ). For example,  $rp(t_e, f_{e,2}) = [v_4, v_1]$  (Fig. 9). 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  $\text{next}(v_4) = rp_2 = v_1$ .

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 subsection). 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.  $f' \leftarrow \text{Transform}(Tr, f, v, q_0, nil, M)$ ;
9. Wait until a transformed fragment is received from each slave site. Let  $f_1, \dots, f_k$  be the received fragments.
10. Merge  $f'$  and  $f_1, \dots, f_k$  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  $(q, pat) \rightarrow h \in R$  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,

$$h(f, Tr, q, v, rp, M) = \begin{cases} h & \text{if } \exists pat((q, pat) \rightarrow h \in R \wedge \text{Eval-}M_{pat}(f, v, pat, rp, M)), \\ \epsilon & \text{otherwise,} \end{cases} \quad (1)$$

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

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

The correctness of the above equation is shown in Section 3.3.  $rp$  and cache  $M$  do not appear in the right-hand side, since they are required for only the evaluation of  $\text{Eval-}M_{pat}$  in a distributed context. As an example of (1), consider the fragment  $f_{e,1}$  shown in Fig. 9 and the tree transducer  $Tr$  of Example 1. Since  $(p, \downarrow^*::\text{branch}) \rightarrow x(pq) \in R$  and  $v_1$  satisfies pattern “ $\downarrow^*::\text{branch}$ ,” we have  $h(f_{e,1}, Tr, p, v_1, nil, M) = x(pq)$ . 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  $h = x(pq)$ , 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**  $\text{child}(f, v) = \emptyset$  **then**
4.   **for** each  $q' \in Q'$  **do**
5.     Replace node  $q'$  in  $h$  with  $\epsilon$ .
6.   **end**
7. **end**
8. Let  $\text{child}(f, v) = \{v'_1, \dots, v'_k\}$ .
9. **for** each  $q' \in Q'$  **do**
10.   **for** each  $v'_i \in \text{child}(f, v)$  **do**
11.      $h_{v'_i} \leftarrow \text{Transform}(Tr, f, v'_i, q', rp, M)$ ;
12.   **end**
13.   Replace node  $q'$  in  $h$  with hedge  $h_{v'_1} \dots h_{v'_k}$ .
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  $rp = [rp_1, \dots, rp_n]$  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

$$Q_r(f, Tr, Q', rp, M) = \{q'' \mid q'' \in St(h(f, Tr, q', rp_n, rp, M)), q' \in Q'\}. \quad (2)$$

Second, if  $n > 1$ , then

$$\begin{aligned} Q_r(f, Tr, Q', rp, M) \\ = \{q'' \mid q'' \in Q_r(f, Tr, Q'', rp', M), \\ Q'' = \{q \mid q \in St(h(f, Tr, q', rp_n, rp, M)), q' \in Q'\}\}, \end{aligned} \quad (3)$$

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

**Example 2** Let  $rp = [v_4, v_1]$  be the root path of  $f_{e,2}$  in Fig. 9,  $Tr = (Q, \Sigma, q_0, R)$  be the tree transducer in Example 1, and  $Q' = \{p\}$ . Consider computing  $Q_r(f_{e,2}, Tr, Q', rp, M)$ . By Eq. (3), we have

$$\begin{aligned} Q_r(f_{e,2}, Tr, Q', rp, M) \\ = \{q'' \mid q'' \in Q_r(f_{e,2}, Tr, Q'', [v_4], M)\}, \end{aligned} \quad (4)$$

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

$$\begin{aligned} Q_r(f_{e,2}, Tr, Q'', [v_4], M) \\ = \{q'' \mid q'' \in St(h(f_{e,2}, Tr, q', v_4, [v_4], M)), q' \in Q''\}, \end{aligned} \quad (5)$$

where  $Q'' = \{p, q\}$ . Since  $(p, \downarrow^*::\text{branch}[\downarrow::\text{location}]/\downarrow::\text{clientele}) \rightarrow y(p)$  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  $h(f_{e,2}, Tr, q, v_4, [v_4], M) = \text{nil}$ . Thus, we have  $Q_r(f_{e,2}, Tr, Q'', [v_4], M) = \{p\}$  by Eq. (5), which implies that  $Q_r(f_{e,2}, Tr, Q', rp, M) = \{p\}$  by Eq. (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.  $f'_q \leftarrow \text{Transform}(Tr, f, v, q', rp, M)$ ;
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 Eq. (1). Thus we define procedures  $\text{Eval-}M_{pat}$  and  $\text{Eval-}M_{pd}$ , which are distributed versions of  $M_{pat}$  and  $M_{pd}$ , respectively. First, we show several definitions. Let  $f$  be a fragment,  $rp = [rp_1, \dots, rp_n]$  be the root path of  $f$ , and  $v$  be a node of  $f$  or  $rp$ . The set of parent nodes of  $v$ , denoted  $\text{parent}'(f, v, rp)$ , is defined as follows.

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

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

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

For example, let  $f_{e,2}$  be the fragment shown in Fig. 9,  $v_7$  be the node in  $f_{e,2}$ , and  $rp = [v_4, v_1]$  be the root path of  $f_{e,2}$ . Then  $\text{Anc}(f_{e,2}, v_7, \downarrow^*, rp) = \{v_5, v_4, v_1\}$ , 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  $res \in \{\text{true}, \text{false}\}$  be the result of the predicate inquiry. Then a cache  $M$  holds a set of pairs  $\langle \text{query}, res \rangle$ , and  $f_M(\text{query})$  denotes the result of *query* to  $M$ , that is,

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

We present procedure  $\text{Eval-}M_{pat}$ . Let  $pat = ls_1 / \dots / ls_n$  be a pattern, where  $ls_i = ax_i :: l_i[pd_{i,1}] \dots [pd_{i,m_i}]$ . 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  $j = 1, \dots, m_n$  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  $pd_{n,1}, \dots, pd_{n,m_n}$  by  $\text{Eval-}M_{pd}$  (line 12,  $\text{Eval-}M_{pd}$  is shown later). When the checks for  $ls_n$  are completed, then the procedure checks if there is an ancestor  $v' \in \text{Anc}(f, v, ax_n, rp)$  of  $v$  such that  $v'$  matches  $ls_1 / \dots / ls_{n-1}$  (lines 21 to 23).

**Procedure**  $\text{Eval-}M_{pat}$

Input : Fragment  $f$ , context node  $v$ , pattern  $pat = ls_1 / \dots / ls_n$  with  $ls_i = ax_i :: l_i[pd_{i,1}] \dots [pd_{i,m_i}]$ , 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  $j = 1, \dots, m_n$  **do**

```

5.  if  $v$  is a node in  $rp$  then
6.    if  $f_M("v + pd_{n,j}") = nil$  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 the fragment stored in  $S$ .
      Let  $res$  be the result.
8.       $M \leftarrow M \cup ("v + pd_{n,j}", res)$ ;
9.    end
10.    $pred\_result \leftarrow f_M("v + pd_{n,j}");$ 
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.   $pat' \leftarrow ls_1 / \dots / ls_n$ ;
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  $pred = ls_1 / \dots / ls_n$  with  $ls_i = ax_i :: l_i[pd_{i,1}] \dots [pd_{i,m_i}]$ , 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 Eq. (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 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.

- $r1: (q, \downarrow :: branch / \downarrow :: currency) \rightarrow h_1$
- $r2: (q, \downarrow :: clientele / \downarrow :: client / \downarrow :: name) \rightarrow h_2$
- $r3: (q, \downarrow :: clientele / \downarrow :: client / \downarrow :: deals) \rightarrow h_3$

From the three patterns above, we construct nested hash functions as shown in Fig. 11. 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. 11. 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  $v'$

```

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

Fig. 11 Nested hash structure constructed from three patterns.

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. 9. Then  $Desc(f_{e,1}, v_1, \downarrow^*, \text{"currency"}) = \{v_2, c_1\}$ .

Let  $pred = ls_1 / \dots / ls_n$ , where  $ls_i = ax_i :: l_i[pd_{i,1}] \dots [pd_{i,m_i}]$ . 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  $pd_{1,1}, \dots, pd_{1,m_1}$ , and
- (b) whether  $v'$  satisfies  $ls_2 / \dots / ls_n$ .

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 sent 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  $pred = ls_1 / \dots / ls_n$  with  $ls_i = ax_i :: l_i[pd_{i,1}] \dots [pd_{i,m_i}]$ , 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  $\lambda(v') = \text{"CONNECT"}$  then
5.       $url \leftarrow \text{attr}(v, \text{"url"})$ ;
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.         $M \leftarrow M \cup ("url + pred", res)$ ;
9.      end
10.      $result \leftarrow result \vee f_M("url + pred")$ ;
11.   else
12.      $p\_res \leftarrow true \wedge \bigwedge_{j=1, \dots, m_1} Eval-M_{pd}(f, v', pd_{1,j}, M)$ ;
13.   if  $n > 1$  and  $p\_res$  then
14.      $pred' \leftarrow ls_2 / \dots / ls_n$ ;
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  $pred = ls_1 / \dots / ls_n$  with  $ls_i = ax_i :: l_i[pd_{i,1}] \dots [pd_{i,m_i}]$ , cache  $M$

Output : none

```

1.  v ← the root node of f;
2.  if ax1 = ↓ then
3.    if l1 = λ(v) then
4.      p_res ← ⋀j=1,⋯, m1 Eval- $M_{pd}(f, v, pd_{1,j}, M)$ ;
5.      if n > 1 and p_res then
6.        pred' ← ls2 / ⋯ / lsn;
7.        res ← Eval- $M_{pd}(f, v, pred', M)$ ;
8.      else if p_res then
9.        res ← true;
10.     end
11.   end
12. else
13.   res ← 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. 9,  $pat = \downarrow^* :: \text{branch}[\downarrow^* :: \text{name}] / \downarrow^* :: \text{currency}$  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  $\downarrow^* :: \text{currency}$ .
  - a. We have  $\lambda(v_2) = \text{“currency”}$  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- $M_{pat}(f_{e,1}, v_1, \downarrow^* :: \text{branch}[\downarrow^* :: \text{name}], nil, M)$`  is called in line 23.
2. By step (1-b) above, check if  $v_1$  matches  $\downarrow^* :: \text{branch}[\downarrow^* :: \text{name}]$ .
  - a. We have  $\lambda(v_1) = \text{“branch”}$  in line 1 of `Eval- $M_{pat}$` . Since  $rp = nil$ , `Eval- $M_{pd}(f_{e,1}, v_1, \downarrow^* :: \text{name}, nil, M)$`  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  $url = \text{“http://server2.com/fe2”}$  and  $pred = \text{“}\downarrow^* :: \text{name.”}$
  - c. If  $f_M(\text{“url + pred”}) = nil$  in line 6,  $S_{e,2}$  is asked to call `pd_inq( $f_{e,2}, \downarrow^* :: \text{name}, M$ )`.
  - d. In  $S_{e,2}$ , `pd_inq( $f_{e,2}, \downarrow^* :: \text{name}, M$ )` 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  $pred = ls_1 / \dots / ls_n$  be a predicate with  $ls_i = ax_i :: l_i[pd_{i,1}] \dots [pd_{i,m_i}]$ . 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.

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

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

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

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  $ns(f) \leq k$ , 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  $ns(f') \leq k$  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  $M_{pd}(t, v, ls) \neq \emptyset$ .  $\square$

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

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

Table 2 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

2.2. If  $ns(f) \leq x$  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  $pred = ls_1 / \dots / ls_n$ , where  $ls_i = ax_i :: l_i[pd_{i,1}] \dots [pd_{i,m_i}]$ . 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  $pd_{1,1}, \dots, pd_{1,m_1}$  (line 12 of  $Eval-M_{pd}$ ) and (b) evaluated with  $pred' = ls_2 / \dots / ls_n$  (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  $j = 1, \dots, m_1$ , (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  $M_{pd}(t, v, pred) \neq \emptyset$ .  $\square$

Let  $pat = ls_1 / \dots / ls_n$  be a pattern with  $ls_i = ax_i :: l_i[pd_{i,1}] \dots [pd_{i,m_i}]$  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  $M_{pat}(t, v, pat) \neq \emptyset$ .

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  $j = 1, \dots, m_n$ ,  $Eval-M_{pd}(f, v, pd_{n,j}, M)$  (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  $M_{pat}(t, v, pat) \neq \emptyset$ .  $\square$

### 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.

- Distributed XPath evaluation algorithms are designed for processing a *single* XPath query efficiently, while an XSLT stylesheet usually contains more than one pattern.
- In general, a pattern used in an XSLT stylesheet is much simpler than an XPath query. **Table 2** shows a summary of the patterns appearing in XSLT 1.0/2.0 W3C recommenda-

tions\*<sup>1</sup> and the examples used in Ref. [19]. 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 Refs. [4], [5], [6] with our method, since no formal computation cost is presented in Refs. [9], [11].

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.

- For each XSLT pattern  $pat$  in  $xs$ , calculate vectors of each node for  $pat$  by using the algorithm.
- 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 than 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 Section 3.2. Hence the parallel computation cost of evaluating the patterns in  $xs$  by our method

\*<sup>1</sup> <http://www.w3.org/TR/xslt/> (XSLT 1.0) and <http://www.w3.org/TR/xslt20/> (XSLT 2.0)

\*<sup>2</sup> This complexity remains the same if  $pat$  uses no predicate and consists of only labels and child axes.

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<sup>\*3</sup> (1.52 GB) is only 7 and that of XMark<sup>\*4</sup> 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.

### 4. Evaluation Experiment

In this section, 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 2.4 GHz Intel Xeon CPU and 4 GB of memory. First, we generated five XML documents of different sizes by XMark [16]. 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 3), while in dataset (B) the root fragment is remarkably heavy (Table 4). Tables 5 and 6 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. 12, 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  $s_0, s_{10}, \dots, s_{100}$ , 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.
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. 13 and 14. For dataset (A), our method is about

Table 3 Sizes of distributed XML documents (dataset A).

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

Table 4 Sizes of distributed XML documents (dataset B).

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

Table 5 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 6 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

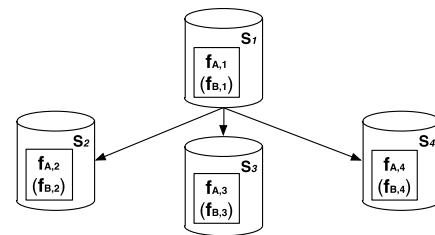


Fig. 12 Four sites storing the fragments in evaluation experiment.

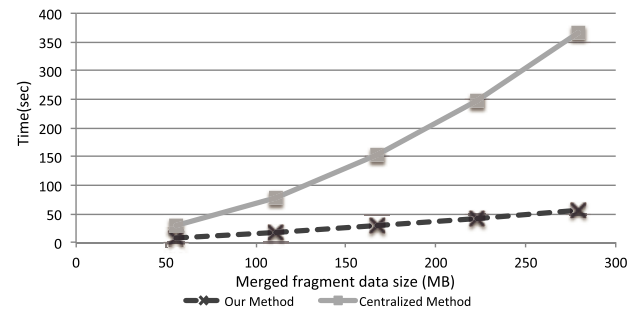


Fig. 13 Experimental result of (a) (dataset A).

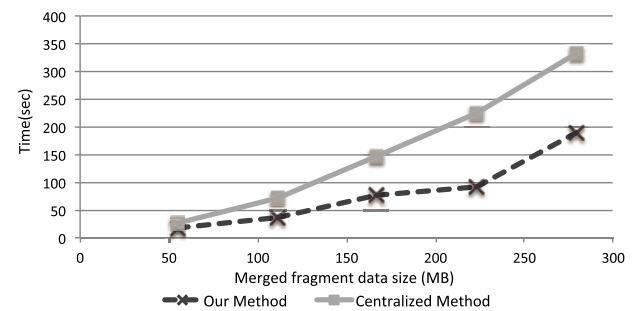


Fig. 14 Experimental result of (a) (dataset B).

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

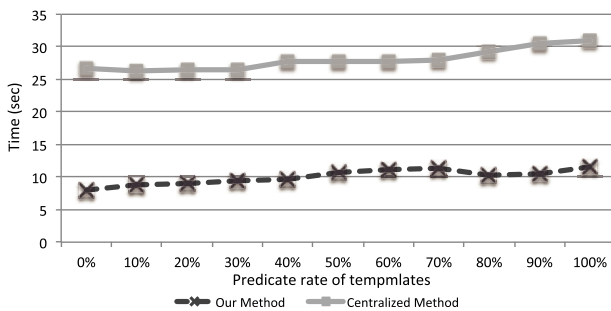
<sup>\*3</sup> <http://dblp.uni-trier.de/xml/dblp.xml.gz>  
<sup>\*4</sup> <http://www.xml-benchmark.org/>

**Table 7** Details of response time for dataset A (sec).

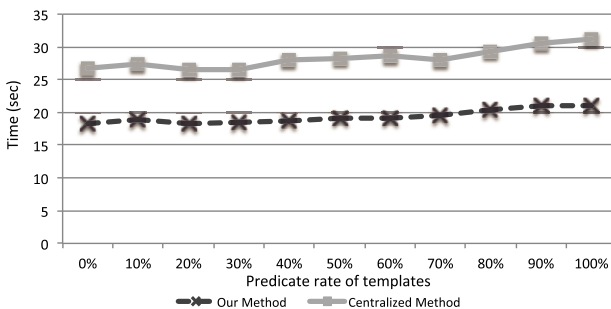
f option (datasize)	Centralized method				Our method
	Transfer	Merge	Transform	Total	
0.5 (55.3 MB)	1.88	0.73	27.04	29.65	7.99
1.0 (111.0 MB)	4.71	1.45	73.04	79.2	17.94
1.5 (167.0 MB)	8.63	2.51	142.87	154.01	28.98
2.0 (222.8 MB)	13.85	2.89	230.06	246.8	41.53
2.5 (279.1 MB)	20.05	3.62	343.42	367.09	56.02

**Table 8** Details of response time for dataset B (sec).

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



**Fig. 15** Experimental result of (b) (dataset A).



**Fig. 16** Experimental result of (b) (dataset B).

is 55.3 MB. **Figures 15 and 16** 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 7 and 8** 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.

## 5. Conclusion

In this paper, 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 paper the expressive power of XSLT is restricted to extended un-ranked top-down tree transducer. In particular, we have to handle XSLT instructions/functions of Type B in Table 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. Finally, we have to investigate the applicability of (non-distributed) XPath processing algorithms to XSLT pattern evaluation. In particular, Ref. [7] proposes a streaming processing algorithm for XPath queries, which constructs a DFA from given XPath filters. Thus the algorithm may be applicable to evaluating XSLT patterns in an XSLT stylesheet, especially if the stylesheet contains a large number of patterns.

**Acknowledgments** The authors would like to thank the editor and anonymous reviewers for their insightful comments to improve this paper. This research is partially supported by Research Center for Knowledge Communities, University of Tsukuba.

## References

- [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, K.: 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] Gupta, A.K. and Suciu, D.: Stream Processing of XPath Queries with Predicates, *Proc. SIGMOD*, pp.419–430 (2003).
- [8] Kepsner, S.: A simple proof for the Turing-completeness of XSLT and XQuery, *Extreme Markup Languages* (2004).
- [9] Kido, K., Amagasa, T. and Kitagawa, H.: Processing XPath Queries in PC-Clusters Using XML Data Partitioning, *Proc. 22nd International Conference on Data Engineering Workshops*, pp.11–16 (2006).
- [10] 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).

- [11] 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).
- [12] Martens, W. and Neven, F.: Typechecking Top-Down Uniform Unranked Tree Transducers, *Proc. ICDT*, Lecture Notes in Computer Science, Vol.2572, pp.64–78, Springer Berlin Heidelberg (2003).
- [13] Mizumoto, H. and Suzuki, N.: A simple XSLT processor for Distributed XML, *Proc. APWEB*, pp.7–18 (2013).
- [14] Mizumoto, H. and Suzuki, N.: An XSLT Transformation Method for Distributed XML, *Proc. INTECH*, pp.71–80 (2014).
- [15] Ogden, P., Thomas, D. and Pietzuch, P.: Scalable XML Query Processing Using Parallel Pushdown Transducers, *PVLDB*, Vol.6, No.14, pp.1738–1749 (2013).
- [16] 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).
- [17] Stefanescu, D.C., Thomo, A. and Thomo, L.: Distributed evaluation of generalized path queries, *Proc. SAC*, pp.610–616 (2005).
- [18] Suciu, D.: Distributed query evaluation on semistructured data, *TODS*, Vol.27, No.1, pp.1–62 (2002).
- [19] Tidwell, D.: *XSLT, 2nd Edition*, O'Reilly Media (2008).
- [20] Zavoral, F. and Dvorakovam, J.: Performance of XSLT processors on large data sets, *Proc. ICADIWT*, pp.110–115 (2009).



**Hiroki Mizumoto** received his bachelor's degree in library and information science from University of Tsukuba in 2013. He has been an M.E. student of Graduate School of Library, Information and Media Studies, University of Tsukuba. His current research interests are distributed XML and tree transducer.



**Nobutaka Suzuki** received his bachelor's degree in information and computer sciences from Osaka University in 1993, and his M.E. and Ph.D. degrees in information science from Nara Institute of Science and Technology in 1995 and 1998, respectively. He was with Okayama Prefectural University as a Research Associate in 1998–2004. In 2004, he joined University of Tsukuba as an Assistant Professor. Since 2009, he has been an Associate Professor of Graduate School of Library, Information and Media Studies, University of Tsukuba. His current research interests include database theory and structured documents.