Incremental Maintenance for Materialized XPath/XSLT Views

Makoto Onizuka
NTT CyberSpace Laboratories, NTT Corporation
1-1 Hikarinooka, Yokosuka, Japan
onizuka.makoto@lab.ntt.co.jp

Fong Yee Chan
Simon Fraser University
Burnaby, B.C, Canada
fyc@sfu.ca

Ryusuke Michigami
Plala Networks Inc.
3-1-15 Ariake, Koto-ku, Tokyo, Japan
mitigami@plala.co.jp

Takashi Honishi
NTT CyberSpace Laboratories, NTT Corporation
1-1 Hikarinooka, Yokosuka, Japan
honishi.takashi@lab.ntt.co.jp

ABSTRACT

This paper proposes an incremental maintenance algorithm that efficiently updates the materialized XPath/XSLT views defined using XPath expressions in $XP^{\{[],*,//,vars\}}$. The algorithm consists of two processes. 1) The dynamic execution flow of an XSLT program is stored as an XT (XML Transformation) tree during the full transformation. 2) In response to a source XML data update, the impacted portions of the XT-tree are identified and maintained by partially re-evaluating the XSLT program. This paper discusses the XPath/XSLT features of incremental view maintenance for subtree insertion/deletion and applies them to the maintenance algorithm. Experiments show that the incremental maintenance algorithm outperforms full XML transformation algorithms by factors of up to 500.

Categories and Subject Descriptors

H.2.3 [**DATABASE MANAGEMENT**]: Languages; D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Optimization*

General Terms

Algorithms, Languages, Performance

Keywords

XML, XPath, XSLT, materialized view, view maintenance

1. INTRODUCTION

As users are demanding more sophisticated Internet services, many web servers are emerging that generate dynamic web pages from databases/files. Examples include DBLP, flight arrival/departure information sites, EPG (electric TV program guide) sites, and stock trading sites. Such systems often use XPath/XSLT processors to transform the source XML data into (X)HTML files even if XPath/XSLT processing is very expensive; a more efficient solution is required.

Two features characterize such dynamic web sites: 1) web pages access is more frequent than source data update. 2) the updated portion is, in each update operation, relatively small compared to the source data size. For example, large numbers of users access the flight arrival/departure information site FlightArrivals.com (http://FlightArrivals.com/) which stores a large amount of data, one day's flight information; flights are updated one by one when each flight's status changes (boarding/expected arrival time/arrived). Therefore, incremental maintenance for materialized XPath/XSLT views is a promising technique to improve web server performance.

Let's discuss below the difficulty of incremental maintenance for materialized XPath and XSLT views.

XPath view maintenance An XPath expression is defined as a sequence of location steps, each of which consists of axis, node-test and optional predicates. The evaluation of each location step returns, from the set of context nodes in XML data, a set of nodes that satisfy the axis relation, node-test and optional predicates. Since XPath permits the use of order-sensitive axes (e.g. following), descendant and other axes, it has higher expression power than SQL. The bad news of these XPath axes evaluations is that, unlike the SQL join evaluation, two nodes (records) are not sufficient to evaluate the axis relation between them. For example, consider the XPath expression //A//C and XML data

```
<A><B><C>NTT Cyberspace Labs.</C></B></A>
```

we name the node whose tag is A as a, B as b, and C as c. The evaluation of the former part (//A) returns a and, using it as a context node, the evaluation of the remainder (//C) returns c. During the descendant axis evaluation of //C, we must access not only a and c but also b that connects a and c. This example suggests that SQL view maintenance techniques [11] are not directly applicable to the XPath view maintenance problem. Here, we have somewhat good news. The labeling scheme of [16, 1], which assigns a *label* to each node, enables us to evaluate all types of axis relations between nodes (e.g. a and c in the above example) without accessing other nodes (e.g. b). Thus by applying the labeling scheme, the axis relation can be implemented by SQL join, so the XPath view maintenance problem is reduced to the SQL view maintenance problem as follows.

Let f be a location step evaluation function, r be a root node in source XML data, D be a set of all nodes, and ls_k ($1 \leq k \leq n$) is a location step of the given XPath expression. The XPath evaluation can then be expressed as follows.

$$f(ls_n, f(ls_{n-1}, ..., f(ls_1, \{r\}, D), .., D), D)$$

WWW 2005, May 10–15, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

Let Δd be a newly inserted set of nodes and D' be $D \cup \Delta d$. Since the location step evaluation function f can be implemented by SQL join with the labeling scheme, we obtain the following expression by applying a differentiation step [11],

$$
\begin{aligned}
& f(ls_n, f(ls_{n-1}, ..., f(ls_1, \{r\}, D'), .., D'), D') = \\
1. \quad & f(ls_n, f(ls_{n-1}, ..., f(ls_1, \{r\}, D), .., D), D) \cup \\
2. \quad & f(ls_n, f(ls_{n-1}, ..., f(ls_1, \{r\}, D), .., D), \Delta d) \cup \\
& ... \cup \\
k. \quad & f(ls_n, ..., f(ls_{n-k+2}, ..., f(ls_1, \{r\}, D), ..., \Delta d), ..., D') \cup \\
& ... \cup \\
n+1. \quad & f(ls_n, f(ls_{n-1}, ..., f(ls_1, \{r\}, \Delta d), .., D'), D')
\end{aligned}
$$

Thus the XPath view can be maintained incrementally by applying the SQL view maintenance techniques.

Unfortunately, the above solution has two problems. First, since it is based on the relational data model, the permitted update operations are node insertion/deletion, which does not efficiently support subtree insertion/deletion, common XML data update operations. Second, it materializes the result of all location steps, so it consumes a huge amount of memory space. For example, consider the following XPath expression to search for *paper*s whose author works in Japan assuming its selectivity is very low.

```
//paper[author/country = "Japan"]
```

Figure 1: An XPath example

This example reveals that, even if the evaluated result of the whole XPath expression is small, the intermediate node-set, which is returned by the location step (//paper) evaluation, becomes quite large (all *paper*s) and can be as large as the source XML data.

XSLT view maintenance The literature on the SQL view maintenance problem [11] categorizes the maintenance techniques from three viewpoints: view language, available data, and modification. We consider the XSLT view problem from the same viewpoints.

In terms of view language, XSLT has higher functionality than SQL, and it can express a transformation that exhibits a loss of structural information, such as removing tags. Consider a materialized view with a loss of structural information and an insertion operation on source XML data. If only the source data and the materialized view are available, it is impossible to identify where to update the materialized view due to the missing structural information. Therefore, from the available data viewpoint, the XSLT view maintenance algorithm requires auxiliary data in general. From the modification viewpoint, we use subtree insertion/deletion, which is permitted by XUpdate [23], and by an XML update language [22].

Our concept is to achieve a space- and time-efficient algorithm to incrementally maintain the materialized views of XPath/XSLT by storing auxiliary data and limiting the XPath to a practical subset $XP^{\{[],*,//,vars\}}$. The algorithm, namely *XTim* (X[ML] T[ransformation] I[ncremental] M[aintenance]) stores the dynamic execution flow of an XSLT program (called XT-tree) which contains the context nodes used by the XSLT templates and a materialized view. *XTim* is space-efficient because it does not store the intermediate node-sets returned by all location step evaluations. *XTim* is also time-efficient because it incrementally maintains a materialized view in three steps: 1) locate the impacted parts in the XT-tree, 2) re-evaluate the XSLT program partially and update the impacted parts, and 3) output the maintained materialized view in the XT-tree. The detail of the first step is as follows. An XT-tree, a dynamic execution flow of an XSLT program, forms an inter-related XPath expression with their context nodes. A single update operation updates a set of updated subtrees and we define *update-path* as the path from the root node to the updated subtree of the source XML data. Then the impacted parts in the XT-tree are identified in a similar way as XML stream processing [7, 10]; it generates an automaton from an inter-related XPath expression and evaluates it on the incoming XML data. But the difference is 1) we identify the impacted XPath expressions in the XT-tree by evaluating the inter-related XPath expression on *update-path*, and 2) we must be aware of the context nodes stored in the XT-tree, because an identical XPath expression can be applied to different context nodes resulting in different XT-nodes in the XT-tree.

1.1 Contributions

Our contributions are summarized as follows:

- We investigate the features of XPath in $XP^{\{[],*,//\}}$ for incremental view maintenance in response to subtree insertion/deletion. In addition, we present the condition under which XSLT expressions inherit the above XPath features and show how to handle those expressions otherwise.
- We develop an incremental view maintenance algorithm *XTim* based on those XPath/XSLT features.
- We discuss the extension of *XTim* to support the ordered data model and position predicates by applying the labeling scheme of [16, 1].
- We describe experiments on typical types of XSLT transformations and the subtree insertion of various sizes. The results show that our algorithm significantly outperforms existing full transformation algorithms by factors of up to 500.

The rest of this paper is as follows. We start by illustrating a motivational example in Section 2. Section 3 defines the fragment of XPath/XSLT specification, the XML update specification, and the incremental view maintenance problem. We investigate the incremental view maintenance features of XPath/XSLT and present *XTim* in Section 4. In Section 5, we discuss how to extend *XTim* to handle the ordered data model. Section 6 reports experimental results. Section 7 addresses related work and Section 8 concludes the paper.

2. MOTIVATIONAL EXAMPLE

We use the author search function at DBLP web site as our example. Indeed the search results for authors are materialized and periodically updated (See the update date of the HTML files in http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/a). Fig. 2 shows a fragment of DBLP XML data (http://dblp.uni-trier.de/xml) and Fig. 3 shows an XSLT program that generates a simplified search result consisting of four XSLT templates. We use only the child axis in the XPath expressions for simplicity. The first template (line 1-9) outputs the given author name and constructs a table for each year in which the author published. The second template (line 10-16) is applied for each year and outputs

```
<dblp>
 <mastersthesis mdate="2002-01-03" key="ms/Brown92">
  <author>Kurt P. Brown</author>
  <title>PRPL: A Database Workload Specification Language,
v1.3.</title>
  <year>1992</year>
  <school>Univ. of Wisconsin-Madison</school>
 </mastersthesis>
 <inproceedings mdate="2002-01-23"
                key="conf/b/Sekerinski98">
 ...
```

Figure 2: DBLP XML data fragment

the year and applies the third template for each publication of the author in the year specified. The third template (line 17-34) constructs a row for the publication. The first column is a link to the electric edition specified by ee tag, if the publication has an electric edition. The second column contains the author list, title, URL, book title, publication year, and page number.

```
 1:<xsl:template match="/">
 2: <html><h1><xsl:value-of select="$author"/></h1>
 3: <table border="1"><tbody>
 4:  <xsl:apply-templates
      select="set:distinct(dblp/*[author=$author]/year)">
 5:   <xsl:sort select="." order="descending"/>
 6:  </xsl:apply-templates>
 7: </tbody></table>
 8:</html>
 9:</xsl:template>

10:<xsl:template match="year">
11: <xsl:variable name="year" select="."/>
12: <tr>
13:  <th colSpan="3"><xsl:value-of select="$year"/></th>
14: </tr>
15: <xsl:apply-templates
      select="/dblp/*[author=$author][year=$year]"
      mode="p"/>
16:</xsl:template>

17:<xsl:template match="*" mode="p">
18: <tr>
19:  <td vAlign="top">
20:   <xsl:if test="ee"><A href="{ee}">EE</A></xsl:if>
21:  </td>
22:  <td>
23:   <xsl:apply-templates select="author"/>:
24:   <xsl:value-of select="title"/>
25:   <A href="http://www.informatik.uni-trier.de/
              ~ley/{url}">
26:    <xsl:value-of select="booktitle"/>
27:    <xsl:text> </xsl:text>
28:    <xsl:value-of select="year"/>
29:   </A>: <xsl:value-of select="pages"/>
30:  </td>
31: </tr>
32:</xsl:template>

33:<xsl:template match="author">
...
```

Figure 3: DBLP author.xsl

Assume some conference is held and several papers are inserted under the dblp tag in DBLP XML data. We need to update the materialized result, however the XSLT full transformation is very expensive because it requires evaluating the XSLT program from scratch. An overview of the incremental maintenance is given below.

1st step The XT-tree is built during the full transformation. Fig. 4 depicts a simplified XT-tree showing only

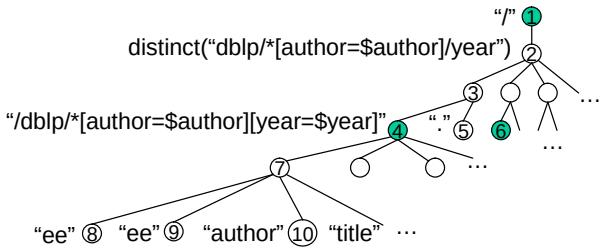


Figure 4: XPath expressions part in XT-tree

the inter-related XPath expression. The gray circles correspond to XT-nodes generated from absolute XPath expressions (line 1,15 in Fig. 3). Since the number of publications attributed to one author is around one hundred, the resulting XT-tree size is kept small.

2nd step Starting from the root XT-node (node 1 in Fig. 4), we process the inter-related XPath expression on the update-path (/dblp). XPath expression "/" of the root XT-node matches to "/" of the update-path then we move to the child XT-node (node 2). The first location step of "dblp/*[author=$author]/year" matches "dblp" of the update-path so we have reached the end of the update-path, i.e. the root node of newly inserted subtree. The remainder part "*[author=$author]/year" is evaluated on the inserted subtree and returns a node-set of years. Since the **distinct** function is used for the node-set, we construct XT-nodes for the resulting year nodes if the XT-tree doesn't store the corresponding XT-nodes. For each of the constructed XT-nodes, we evaluate the second XSLT template and continue re-evaluating the XSLT program partially.

3rd step For each XT-node (node 4,6,...) generated from the absolute XPath expressions, we follow the same procedure as for the root XT-node. The first localtion step of /dblp/*[author=$author][year=$year] matches the update-path, so we have reached the root node of inserted subtree and the remainder part *[author=$author][year=$year] is evaluated on the inserted subtree. We then continue re-evaluating the XSLT program partially.

4th step The XT-tree has been maintained for the update operation, so we output the materialized view stored in the XT-tree.

3. PREMISE

3.1 XPath

$XP^{\{[],*,//,vars\}}$ denotes the XPath fragment that permits predicate, wildcard, child and descendant axis, and variable references. $XP^{\{[],*,//\}}$ consists of expressions given by the following grammar:

$$
\begin{aligned}
P &::= P\ '|'\ P \mid P_{absolute} \mid P_{relative} \\
P_{absolute} &::= {}'/'P_{relative} \\
P_{relative} &::= step\ ('/'step)* \\
step &::= axis\ '::'\ node{-}test\ ('['predicate']')* \\
axis &::= child \mid descendant \\
node{-}test &::= name \mid @name \mid * \mid @* \mid text'()' \\
predicate &::= P \mid general\ predicate
\end{aligned}
$$

XPath expression (XPE) P can have disjunction (|) and be either an absolute or a relative expression. A relative expression is a sequence of steps, each of which consists of axis,

node-test, and optional predicates. The XPath functions [6] can be used in a *general predicate*.

Although $XP^{\{[],*,//,vars\}}$ doesn't permit the use of order-sensitive axes, it is practical for general data-oriented XML data; Typical data-oriented XML data, including the meta data for TV program guides (MPEG7 [13], TVAnyTime [24], P/META [20]) or digitized medical records, are not sensitive to node order. Thus, order-sensitive axes are not used for those XML data. In addition, we assume reverse axes (parent, ancestor) are rewritten to forward axes using XPath rewrite rules [18].

3.2 XSLT

We simplify XSLT 1.0 [5] from two viewpoints: functionality of XML transformations and re-writability to other equivalent XSLT expressions.

First from the viewpoint of functionality, we do not consider the following XSLT expressions since they are not essential for XML transformation: modularization (xsl:apply-import, xsl:import, xsl:include), output formatting (xsl:output, xsl:preserve-space, xsl:processing-instruction, xsl:strip-space, xsl:decimal-format,xsl:number), and other functions (xsl:fallback, xsl:message, xsl:namespace-alias).

Second from the viewpoint of re-writability, we do not consider the following XSLT expressions, since they can be re-written using equivalent and more fundamental XSLT expressions. xsl:call-template, xsl:for-each can be re-written to xsl:apply-templates with parameters and **mode**. xsl:when, xsl:otherwise, xsl:choose are equivalent to a set of xsl:if. xsl:attribute-set is equivalent to a set of xsl:attribute. xsl:key is equivalent to an equi-join operation expressed by a predicate of an XPE. In addition, the match pattern of XSLT template (Remember *match* and *select* pattern are different in XSLT) is simplified to permit the use of current node test, because an XSLT template whose match pattern uses a path expression can be re-written to two XSLT templates whose match pattern uses a current node test. The rewriting is as follows.

1. The first XSLT template uses the node-test of the first step of the original XPE as the match pattern. If the first step has predicates, they are re-written to the xsl:if condition. The remaining part of the original XPE is used as the select pattern of an xsl:apply-templates.

2. The second XSLT template uses the node-test of the last step of the original XPE as the match pattern.

3. The above two XSLT templates are connected using the mode of xsl:template and xsl:apply-templates.

For example, an XSLT template whose match pattern is A[B][C]/D[E]//F is re-written to the following two XSLT templates.

```
<xsl:template match="A">
  <xsl:if test="(.)[B][C]">
    <xsl:apply-templates select="D[E]//F" mode="S1"/>
  </xsl:if>
</xsl:template>
<xsl:template match="F" mode="S1">
  (the content of the original XSLT template)
</xsl:template>
```

Our simplified XSLT also permits XPEs to use variable references ($XP^{\{[],*,//,vars\}}$) and so is a more general specification of the XSLT fragment $XSLT_0$ [4]. Appendix A of the full paper [19] shows the syntax of the simplified XSLT.

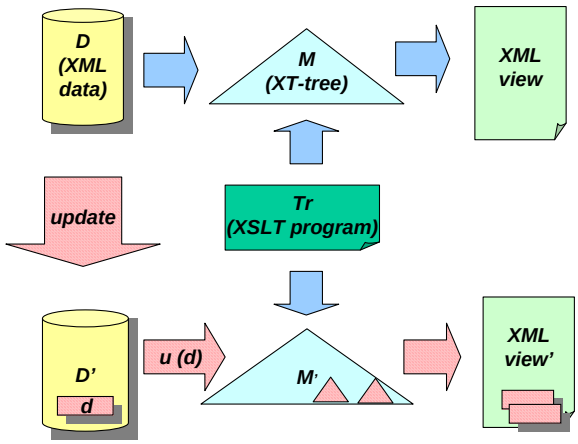


Figure 5: Incremental view maintenance

3.3 XUpdate

We modified the XUpdate specification [23] to express how the source data was updated[1]. **insert(*p*,*r*)** expresses the inserted subtree r and its path p (called *update-path*). Symmetrically **delete(*p*,*r*)** expresses the deleted subtree r and its path p. The *update-path* uniquely identifies a path from the root node of source XML data to the root node of the updated subtree and consists of a sequence of *update steps*, each of which is pair of nodeID and node name.

For example, Fig. 6 expresses an update expression in which the third *author* is inserted for the *paper* whose nodeID is 5.

```
insert(/(1,bib)/(5,paper)/(8,author),
       "<author role="3rd">
          <name>makoto onizuka</name>
          <country>Japan</country>
        </author>"
```

Figure 6: An XUpdate example

3.4 Problem definition

Fig. 5 depicts the problem of incremental maintenance for materialized XPath/XSLT views which we express as follows: "Let D be source XML data, $D^{'}$ be the modified source XML data, Tr be an XSLT program, $t(D, Tr)$ be a function that evaluates Tr on D, $u(d)$ be an XUpdate expression describing how the source data was updated[2], and M be auxiliary data constructed from D and Tr beforehand. Our goal is to output $t(D^{'}, Tr)$ by incrementally maintaining M for update expression $u(d)$."

4. INCREMENTAL VIEW MAINTENANCE

This section investigates the XPath/XSLT features that support incremental view maintenance (Section 4.1,4.2), then describes the auxiliary data XT-tree (Section 4.3) and the incremental view maintenance algorithm *XTim* (Section 4.4).

[1]When multiple parts are updated, each part is expressed by our modified XUpdate expression.
[2]d is an updated subtree.

$$eval(\{p\}, xp_s, p) \rightarrow match(xp_s, p) \quad (1)$$

$$match(xp_s, p) \rightarrow \begin{cases} true & \text{if both } xp_s \text{ and } p \text{ are empty strings} \\ false & \text{if only } xp_s \text{ or } p \text{ is empty string} \end{cases} \quad (2)$$

$$match(xp_1|...|xp_n, p) \rightarrow match(xp_1, p) \vee ... \vee match(xp_n, p) \quad (3)$$

$$match(child :: N/R_{xp}, H/R) \rightarrow \begin{cases} match(R_{xp}, R) & \text{if } N \text{ matches } H \\ false & \text{otherwise} \end{cases} \quad (4)$$

$$match(descendant :: N/R_{xp}, H/R) \rightarrow \begin{cases} match(descendant :: N/R_{xp}, R) \vee match(R_{xp}, R) & \text{if } N \text{ matches } H \\ match(descendant :: N/R_{xp}, R) & \text{otherwise} \end{cases} \quad (5)$$

Figure 7: *eval* algorithm (XPath expression is in $XP^{\{*,//\}}$)

To identify the impacted XPath expressions in an XT-tree by update operations, we consider the XPath semantics that determines if a path matches an XPath expression. Thus, our semantics help to identify the impacted XPath expressions by using a set of paths to nodes in an updated subtree. We can easily extend the semantics to an inter-related XPath expression and *XTim* implements the extended semantics.

4.1 XPath expression

We define P as a set of paths from the root node to every node in the source XML data. We define the XPath evaluation algorithm *eval* on a single path.

Definition 4.1 *$eval(P, xp, p)$ is a Boolean function that decides if the given XPath expression, xp, matches path p. The first parameter, P, is the scope that the **eval** function may refer to during the xp evaluation on p.* ∎

Accordingly, we define the XPath evaluation function $eval_c$ on a set of paths.

Definition 4.2 *Let P' be a subset of P, and xp be an XPath expression. We define the XPath evaluation function on P' as:*

$$eval_c(P, xp, P') = \{p \mid p \in P', eval(P, xp, p)\}$$ ∎

Thus the evaluation of xp on P, all paths in the source XML data, is expressed by $eval_c(P, xp, P)$.

Let's consider XPath expressions without predicates first and extend the coverage later. If xp_s is in $XP^{\{*,//\}}$, then $eval(P, xp_s, p)$ can be implemented by finite automata [7, 10] without referring to other paths in P, since xp_s is a regular expression and p is a sequence of nodes with a name. Therefore, $eval(P, xp_s, p) = eval(\{p\}, xp_s, p)$ and we have the following result.

Proposition 4.3 *Let xp_s be in $XP^{\{*,//\}}$ and P' be a subset of P. The $eval_c$ function on P' is evaluated without reference to other paths in P.*

$$eval_c(P, xp, P') = eval_c(P', xp, P')$$ ∎

Fig. 7 shows an algorithm of the $eval(\{p\}, xp_s, p)$ function when $xp_s \in XP^{\{*,//\}}$. Step (1) shows $eval(\{p\}, xp_s, p)$ implemented by $match(xp_s, p)$. Step (2) is a termination rule and steps (3)-(5) are recursive rules. Step (3) shows the rule to be applied when xp is a disjunctive expression. If one of $match(xp_i, p)$ $(1 \leq i \leq n)$ is true, $match(xp_1|...|xp_n, p)$ returns true. Step (4) shows the rule to be applied when the first step axis of given XPE is *child*. If the node-test N of the first step of the XPE matches the first step H of the given path, then continue to check that the remaining part of the XPE R_{xp} matches the remaining part R of the path. Otherwise, the ***match*** function fails. Step (5) shows the rule to be applied when the first step axis of XPE is *descendant*. If the node-test N of the first step of the XPE matches the first step H of the given path, then non-deterministically continue to check that R is matched by the same XPE or the remaining part R_{xp}. Otherwise, continue to check that R is matched by the same XPE. After applying the ***match*** function recursively, one of the two parameters of the ***match*** becomes an empty string. Step (2) shows the rule to be applied if either of the parameters is empty. If both of them are empty then the ***match*** returns *true*, otherwise if only one them is empty then the ***match*** returns *false*.

Now consider XPath expressions with predicates and the scope on source XML data that may be referred to during view maintenance. Note *update-path* of an XUpdate expression is the common prefix of paths to nodes in the updated subtree. We exploit *update-path* to share the process of identifying impacted XPath expressions for each path.

Theorem 4.4 (scope) *Let xp be in $XP^{\{[],*,//\}}$, ΔP be the set of paths to nodes in the updated subtree, and $step_{xp}$ be the top most step with predicate in xp, and $step_{update}$ be the top most update step in update-path that the node-test of $step_{xp}$ matches. If some path in ΔP matches the last location step in xp or some predicate in xp, then the **eval** algorithm on ΔP is evaluated with reference to all paths (P") that contain the node indicated by $step_{update}$.*

$$eval(P, xp, \Delta P) = eval(P'', xp, \Delta P)$$ ∎

PROOF. (sketch) *update-path* and XPath xp are processed by the *eval* algorithm in Fig. 7 until $step_{update}$, because there is no step in xp with predicate until $step_{xp}$. The remaining part of xp starting from $step_{xp}$ can then be evaluated on the subtree whose root node is indicated by $step_{update}$, because the permitted child and descendant axes refer only to the subtree. □

For example, assume that the update operation expressed in Fig. 6 makes the *paper* whose nodeID is 5 match the XPath expression in Fig. 1. In this case, $step_{xp}$ is //paper and $step_{update}$ is (5,paper), so P" becomes all paths in such subtree whose root nodeID is 5. Therefore, we re-evaluate //paper[author/country = "Japan"] on the node whose nodeID is 5 without referring to the remaining part of the source XML data.

While Theorem 4.4 explains the scope on source during view maintenance, an update operation does not always impact materialized nodes.

Theorem 4.5 (unaffected) *Let xp and ΔP be the same as above, n_{mat} be a materialized node of xp before the update operation was commenced. If a step $step_{update}$ in update-path corresponds to n_{mat} and none of the paths in ΔP match any predicate in xp, then the update operation does not impact n_{mat}.* ∎

PROOF. (sketch) *update-path* is processed until $step_{update}$ by the *eval* algorithm in Fig. 7 using XPath xp, because none of the paths in ΔP match any predicate in xp. The update operation does not impact n_{mat}, because 1) the materialization of n_{mat} indicates all predicates in xp were evaluated as true, and that 2) none of the paths in ΔP match any predicate in xp. □

For example, assume node n whose nodeID is 5 is materialized by //paper[year>2002] and the update operation expressed in Fig. 6 is applied to n. Theorem 4.5 is applicable to this case, so we don't need to re-evaluate the XPath expression.

4.2 XSLT expressions

We present the condition under which XSLT expressions inherit the XPath features described in Section 4.1 and how to handle those expressions otherwise. There are three types of XPath usage in XSLT expressions: xsl:apply-templates, xsl:if, view construction as xsl:value-of, and xsl:copy-of. First we consider XSLT expressions whose XPath expression does not use variable/parameter; we then show how to handle variable/parameter.

xsl:apply-templates The evaluation of xsl:apply-templates's XPE without xsl:sort inherits the XPath features, so we need to materialize just the node-set returned by the XPath evaluation. When xsl:sort is specified, we need to consider how to maintain the sorted node-set efficiently. Our approach is to materialize the sorted pairs (node, key value). When an update operation impacts the evaluation of xsl:apply-templates's XPE, we identify the position of the impacted pair in the sorted pairs by binary search, and then insert/delete the pair to/from the position indicated.[3]

xsl:if Since xsl:if is evaluated with existential quantification, it is not sufficient for incremental view maintenance to materialize the XPath result. We apply the **counting** algorithm [12], which was originally designed for the incremental maintenance of SQL views with set semantics, to the xsl:if evaluation and store the number of nodes that satisfy the xsl:if condition. When an update operation impacts the xsl:if condition evaluation, we add/subtract the number of updated nodes that satisfy/do not satisfy the xsl:if condition. If the resulting number becomes zero (indicating xsl:if condition is evaluated as false), we delete the materialized view of the child XSLT expression. If the resulting number that was originally zero becomes one (indicating xsl:if condition is evaluated as true), we add the materialized view of the child XSLT expression.

view construction The evaluation of the view construction's XPE inherits the XPath features, thus we need to materialize just the result.

xsl:variable/param xsl:variable/param inherits XPath features. Fig. 8 shows an example wherein variables $V_1 ... V_n$ are defined by referring to the previous variable. Since the bound value can be seen as the input XML data for the next variable, the final variable V_n is incrementally maintained in the same way as described above by generating the XUpdate expressions ($u(d_1)$,...,$u(d_{n-1})$) for the bound value.

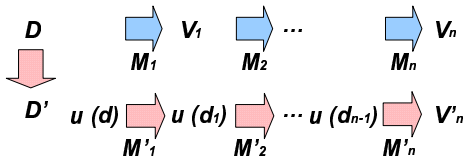


Figure 8: Variable maintenance

4.3 XT-tree

We define the auxiliary data XT-tree for incremental view maintenance of an XSLT program, a set of XSLT expressions.

4.3.1 XT-tree structure

An XT-tree is a tree of XT-nodes each of which contains a reference to the XSLT expression it was generated from. An non-leaf XT-node commonly stores a sequence of references to child XT-nodes that expresses the dynamic execution sequence of XSLT expressions.[4] There are five types of XT-node: XT-template, XT-param, XT-if, XT-node-set, and XT-view, each of which is constructed from its corresponding XSLT expression.

XT-template: An XT-template instance stores its context node.

XT-param An XT-param instance is constructed for either xsl:param or xsl:variable and stores the bound value.

XT-if An XT-if instance is constructed for xsl:if. When the xsl:if condition is satisfied, a child XT-node is constructed by evaluating the child XSLT expressions of xsl:if.

XT-node-set An XT-node-set instance is constructed for xsl:apply-templates and stores a set of references to child XT-templates that are constructed by the applied XSLT templates. The set of references are sorted by the context nodeID of the child XT-templates so as to identify, by binary search, the node position for insertion/deletion. When xsl:sort expressions are specified, XT-node-set stores a list of pairs (*reference to child XT-node, key value*) sorted by the *key value* in the specified order (descending/ascending). The node position for insertion/deletion is identified by binary search using the node's key value and nodeID.

XT-view An XT-view instance stores a materialized view and is constructed for xsl:element, xsl:attribute, xsl:text, xsl:value-of, xsl:copy, or xsl:copy-of.

For example in Fig. 9, the left part shows the source XML data and the middle part shows the XT-tree constructed by the XSLT program in the right part. The XT-tree indicates that the four *section* nodes (depicted by the gray circles) are used as the context of the second XSLT template.

4.3.2 XT-tree construction

The XT-tree is built during the full transformation. In addition, we build an XT-node-set *candidates* to store all XT-nodes, including the root XT-node, generated from the XSLT expression with an absolute XPE. The optimized incremental view maintenance in Section 4.4 utilizes *candi-*

[3]There is an XSLT extension library that supports the **distinct** operation to remove duplicated values in a node-set. We can apply the **counting** algorithm in the same way as xsl:if processing.

[4]An XT-tree contains the applied default templates.

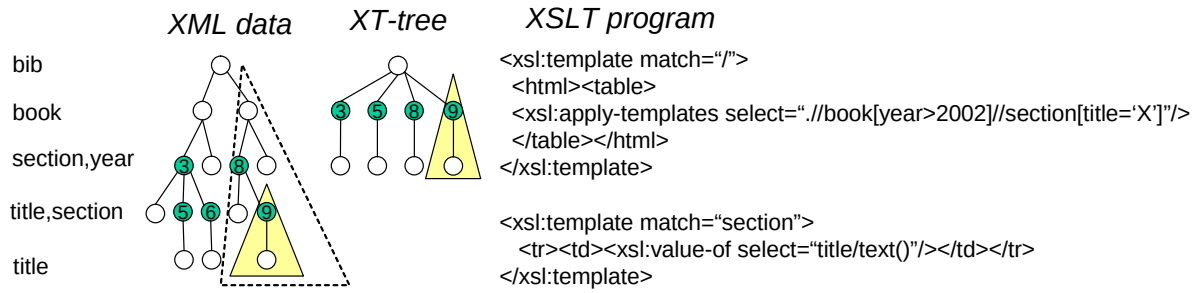


Figure 9: Incremental view maintenance example

dates to avoid traversing XT-nodes that are not impacted by update operations.

4.4 Incremental view maintenance

Note that an XT-tree, a tree of XSLT expressions, forms an inter-related XPath expression. *XTim* extends the XPath semantics in Section 4.1 so as to determine the impacted XPath expressions in the inter-related XPath expression by update operations.

The view maintenance process consists of three steps: 1) identify the impacted XPath expressions for *update-path* and locate the context node in the source XML data, 2) re-evaluate the XSLT program partially on the located context node and maintain the impacted XT-nodes, and 3) output the maintained materialized view stored in the XT-tree. During the 1st step, for each *update-step* in *update-path* *XTim* uses Fig. 7 to process XPath expressions. When some XPath expression becomes empty, *XTim* locates a child XT-node whose context node is identical to *update-step* and continues processing the XPath expression of the child XT-node. When *XTim* reaches to the end of *update-path*, it reaches the root nodes of *impacted XT-nodes* (Fig. 10) and moves to the 2nd step. In the 2nd step, *XTim* evaluates the XPath expression on the context node specified by *update-step* and maintains the XT-tree. After the 2nd step, since the XT-tree stores the maintained materialized view, the 3rd step is easy to complete.

We focus on the XT-tree maintenance algorithm for an XUpdate *insert* expression, since the symmetry of XUpdate *insert* and *delete* expressions makes the maintenance algorithm symmetric. One naive method of XT-tree maintenance is to traverse all the XT-nodes in the tree to locate XT-nodes generated from XSLT expressions with absolute XPath expressions. This method is sound because it ensures that all impacted XT-nodes are found and that the referred variables always bind the maintained values. However, since it may not be efficient to traverse all the XT-nodes in the XT-tree, we propose an optimized method that avoids traversing all XT-nodes. This method utilizes *candidates* which store all XT-nodes, including the root XT-node, with absolute XPath expressions. This approach is also sound because it ensures that all impacted XT-nodes are found by using the *candidates* and that the variable is lazily maintained when its bound value is referred to.

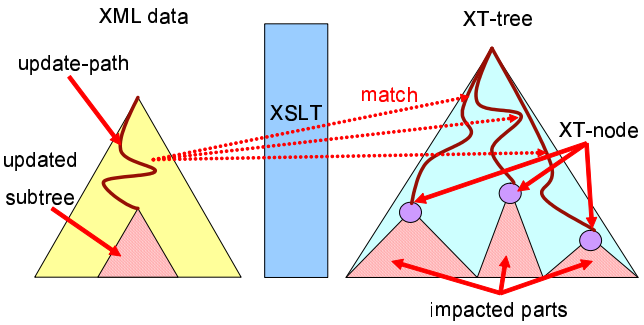


Figure 10: *XTim* view maintenance

```
Algorithm maintain-XT-tree(D,u-org,candidates)
Input:  D is the updated source XML data.
        u-org is the original XUpdate expression.
        candidates is a set of XT nodes.
1. r = get-document-root(D);
2. for-each XT-node in candidates
3.   if is-maintained(XT-node) continue;
4.   else maintain-XT-node(XT-node,u-org);
```

Figure 11: maintain-XT-tree

```
Algorithm maintain-XT-node(p-XT, u)
Input:  p-XT is current XT-node.
        u is an XUpdate expression.
1. switch (get-XT-node-type(p-XT))
2.   case XT-template:
3.     break;   // continue to line 18
4.   case XT-param:
5.     return;  // deferred evaluation
6.   case XT-node-set:
7.     proceed-step(p-XT,u,get-xpath(p-XT));
8.     return;
9.   case XT-if:
10.    for-each xpath in get-xpaths(p-XT)
11.      proceed-step(p-XT,u,xpath);
12.    // end for-each
13.    return;
14.  case XT-view:
15.    proceed-step(p-XT,u,get-xpath(p-XT));
16.    break;   // continue to line 18
17. // end switch
18. for-each c-XT in get-children-XT-nodes(p-XT)
19.   if is-maintained(c-XT) continue;
20.   else maintain-XT-node(c-XT,u);
```

Figure 12: maintain-XT-node

4.4.1 Algorithm

Fig. 11 shows the main function. It parses the updated source XML data D to allow the re-evaluation of some predicate of XPath expressions to access portions of D (line 1). For each *XT-node* in *candidates* (line 2), if it has already been maintained then skip to the next XT-node to avoid maintaining the same XT-node twice (line 3). Otherwise, it invokes the **maintain-XT-node** function for current *XT-node* (line 4).

Fig. 12 shows the **maintain-XT-node** function that maintains the current XT-node *p-XT* according to its type. When

```
Algorithm proceed-step(XT-node, u, or-xpath)
Input:  XT-node is current XT-node.
        u is an XUpdate expression for or-xpath.
        or-xpath is an XPath expression.
 1. or-xpath-next = null;
 2. u-path = get-update-path(u);
 3. u-step = get-first-step(u-path);
 4. for-each xpe in get-paths-from-or-expr(or-xpath)
 7.  t-step = get-first-step(xpe);
 8.  switch (get-axis-type(t-step))
16.   case descendant:
17.    if match-node-test(u-step,t-step)
18.     if is-predicate-impacted(u,t-step)
19.      update-XT-node(XT-node,get-node(u-step),xpe);
20.      break;
21.     else or-xpath-next = or-expr(or-xpath-next,
                                     get-rest-path(xpe),
                                     xpe);
22.    else or-xpath-next = or-expr(or-xpath-next,xpe);
24.  // end switch
26. // end for-each
27. if (XPath match fails) return;
29. if (reached the updated subtree)
30.   update-XT-node(XT-node,get-node(u-step),
                     or-xpath-next);
31.   return;
33. if (A part of XPath is matched)
35.   proceed-XT-node(XT-node,u-next,get-node(u-step));
36. proceed-step(XT-node,u-next,or-xpath-next);
```

Figure 13: proceed-step

p-XT's type is XT-template indicating that it was generated from an xsl:template expression (lines 2-3,18-20), all child XT-nodes of *p-XT* are to be maintained: for each child XT-node *c-XT* of *p-XT*, if it has already been maintained then skip to the next XT-node (line 19), otherwise the **maintain-XT-node** function is invoked recursively for *c-XT* (line 20). When *p-XT*'s type is XT-param (lines 4-5), maintenance is deferred until the variable/parameter is referred to. When *p-XT*'s type is XT-node-set (lines 6-8), the **proceed-step** function is invoked to check matching between the XPE of the corresponding XSLT expression and *update-path* of the XUpdate expression, and to maintain the descending part of *p-XT*. When *p-XT*'s type is XT-if (lines 9-13,18-20), the **proceed-step** function is invoked for all XPEs used in the xsl:if condition and to maintain the descending part of *p-XT*. If *p-XT*'s type is XT-view (lines 14-16,18-20), when the corresponding XSLT expression is either xsl:value-of or xsl:copy-of that uses an XPE, the **proceed-step** function is invoked. The function then processes all child XT-nodes of *p-XT* (lines 18-20).

Fig. 13 shows the key steps of the **proceed-step** function[5]; Appendix B of the full paper [19] shows the complete program. The **proceed-step** function processes the first update step (*u-step*) in *update-path* and is recursively invoked (line 36) until it reaches one of three cases; 1) XPath match fails (line 27), 2) all *update step*s in *update-path* have been processed, indicating that we have reached the root node of the update subtree in the source XML data (lines 29-31), or 3) a part of XPath is matched (lines 33-36). In the 1st case, the XUpdate expression doesn't impact the current *XT-node*. In the 2nd case, it invokes the **update-XT-node** function to update the current *XT-node* by evaluating *or-xpath-next*. Theorem 4.4 explains the scope on the source XML data during the processing of the **update-XT-node** function. In the 3rd case, there are two tasks; one is to continue processing the unmatched part of XPath (line 36). The other is to invoke the **proceed-XT-node** function for locating a child XT-node and then traversing the XT-tree. The **proceed-XT-node** function implements Theorem 4.5 and locates the child XT-template whose context nodeID is identical to the nodeID of *u-step*.

In general, since the XPath expression *or-path* is disjunctive, the **proceed-step** function processes each subexpression *xpe* in *or-path* (lines 4-26). *t-step* is the first step of *xpe* (line 7). If the node-test of *t-step* matches the first step of *u-path* (line 17), it continues to check that the predicates of *t-step* are impacted by the XUpdate expression (line 18). Otherwise (line 22), the **proceed-step** function sets *or-xpath-next*, which corresponds to the 2nd line in (5) of the **eval** algorithm in Fig. 7. If some predicate of the first step (*t-step*) of *xpe* is impacted by the XUpdate expression, the **update-XT-node** function is invoked to update current *XT-node* (line 19). Otherwise (line 21), the **proceed-step** function sets *or-xpath-next*, which corresponds to the 1st line in (5) of the **eval** algorithm.

4.4.2 Example

Consider the example depicted in Fig. 9. The small triangle in the left part depicts a new subtree inserted into the source XML data. The update-path is expressed as /(bib,1)/(book,2)/(section,8)/(section,9). The triangle in the middle part shows the expected new subtree in the XT-tree. There are two select XPath expressions in the XSLT program; .//book[year>2002]//section[title='X'] (xp_1) and title/text() (xp_2). Incremental maintenance is done as follows.

1st step Starting from the root node of the XT-tree, (bib,1) is processed by xp_1 and the XPE for the next processing (*or-xpath-next*) is also xp_1 (line 22 in Fig. 13). Then (book,2) is processed and *or-xpath-next* becomes xp_1|//section[title='X'] (line 21 in Fig. 13). Finally, (section,8) is processed and *or-xpath-next* becomes xp_1|//section[title='X']| (*null*). The (*null*) indicates partial XPath matching, so p1) we continue processing the unmatched part of XPath, and p2) we locate the child XT-node whose context nodeID is identical with the current update step (section,8) and continue traversing the XT-tree.

2nd step Now we have reached the node whose nodeID is 9, which is the root node of the updated subtree. For case p1 (current XT-node is the root), we evaluate $xp1$|//section[title='X'] on the XML node whose nodeID is 9. The scope on the source XML data is depicted by the triangle consisting of dotted lines in Fig. 9, because we need to evaluate the predicate [year>2002] whose evaluation we have skipped up to now. The remaining process is the same as that part of the full transformation process; construct new XT-nodes and continue processing the second XSLT template. For case p2, the located XT-node whose nodeID is 8 is constructed from the second XSLT template, so we evaluate $xp2$ on the XML node whose nodeID is 9. This case doesn't impact the XT-tree.

5. ORDERED DATA MODEL

Since *XTim* stores a node-set sorted by nodeID and the labeling schemes [16, 1] can encode the node position into the nodeID (*label*), the labeling schemes enable *XTim* to support the ordered data model. Future work includes de-

[5]Thus the line number skips.

	depth	*data size (KB)*	*# of elements*
D7	7	141	3280
D8	8	464	9841
D9	9	1527	29524
D10	10	4949	88573
D11	11	16067	265720

(a) XML data

	description	*selectivity*
simple	preserving structure	7.8%
descendant	flatten structure	66.7%
sort	**descendant** + sort	66.7%
simple-pred	**simple** + predicate	1.0%
descendant-pred	**descendant** + predicate	2.5%

(b) XSLT programs

Table 1: XML data & XSLT programs

termining how to update the nodeIDs stored in the XT-tree when nodeIDs in the source XML data are re-labeled.

There are two approaches to maintaining the evaluation result of XPath with a position predicate; 1) re-evaluation, and 2) materialization. Consider //book[year>2002][3]/title, which extracts the *title* of the third *book* published after 2002. When a newly inserted *book* has a publication year of 2003, the position predicate may be impacted by the update operation, thus we need to maintain the materialized XPath view. The re-evaluation approach evaluates the XPath expression fully, so it is not efficient w.r.t. speed but it doesn't require additional memory space. The materialization approach materializes the node-set returned by //book[year>2002], incrementally maintains the node-set, and evaluates [3] on the materialized node-set. This is efficient w.r.t. speed but requires additional memory space.

6. EXPERIMENTS

We implemented a persistent DOM manager (PDOM) and *XTim* using the XML parser Xerces-J 2.6.2 and the XPath processor Jaxen 1.1. The PDOM loads XML files and assigns a persistent nodeID to each node. Since our data model is not node order sensitive, a new node receives a larger nodeID than the existing nodes. The PDOM receives an original XUpdate expression [23], updates the stored DOM, and submits rewritten XUpdate expressions (defined in Section 3.3) to the *XTim* algorithm. Our execution environment consisted of an Intel Pentium M 1.6GHz PC with 2048MB main memory, running Windows XP, and Sun Java SDK 1.4.2_06. For each experiment, the performance result doesn't include the DOM building time nor the XSLT program parsing time.

XML data: We used nested synthetic XML data that forms a balanced 3-ranked tree as shown in (a) of Table 1. We also conducted experiments using DBLP XML data, a typical example of shallow XML. We omit the latter here, since the result is similar to that of the former synthetic XML data.

XSLT programs: We used four different types of XSLT programs as shown in (b) of Table 1. **simple** represents a structure preserving transformation that constructs a tree whose size is 7.8% of the input XML data. **descendant** represents a flatten-structure transformation using descendant axis and constructs a tree whose size is 66.7% of the input XML data. **sort** represents a flatten-structure transformation with sort. **simple-pred** and **descendant-pred** are **simple** with a predicate and **descendant-pred** with a predicate respectively.

XML update operations: We used four different subtree sizes for insertion: 4KB, 14KB, 43KB, and 141KB. They are also balanced 3-ranked trees with depths of 4, 5, 6, and 7 respectively.

6.1 XT-tree building

Fig. 14 shows XT-tree size and the response time of XT-tree building. (a) indicates that XT-tree size (the number of nodes) scales reasonably to source XML size (the number of nodes) and the selectivity of the XSLT program. The combination of (a) and (b) indicates that the XT-tree building performance depends on the XT-tree size in general. However, it is not true for **descendant-pred** transformation, because the evaluation of descendant axis requires access to all the nodes in the source XML data, which is expensive.

6.2 Comparison with SAXON

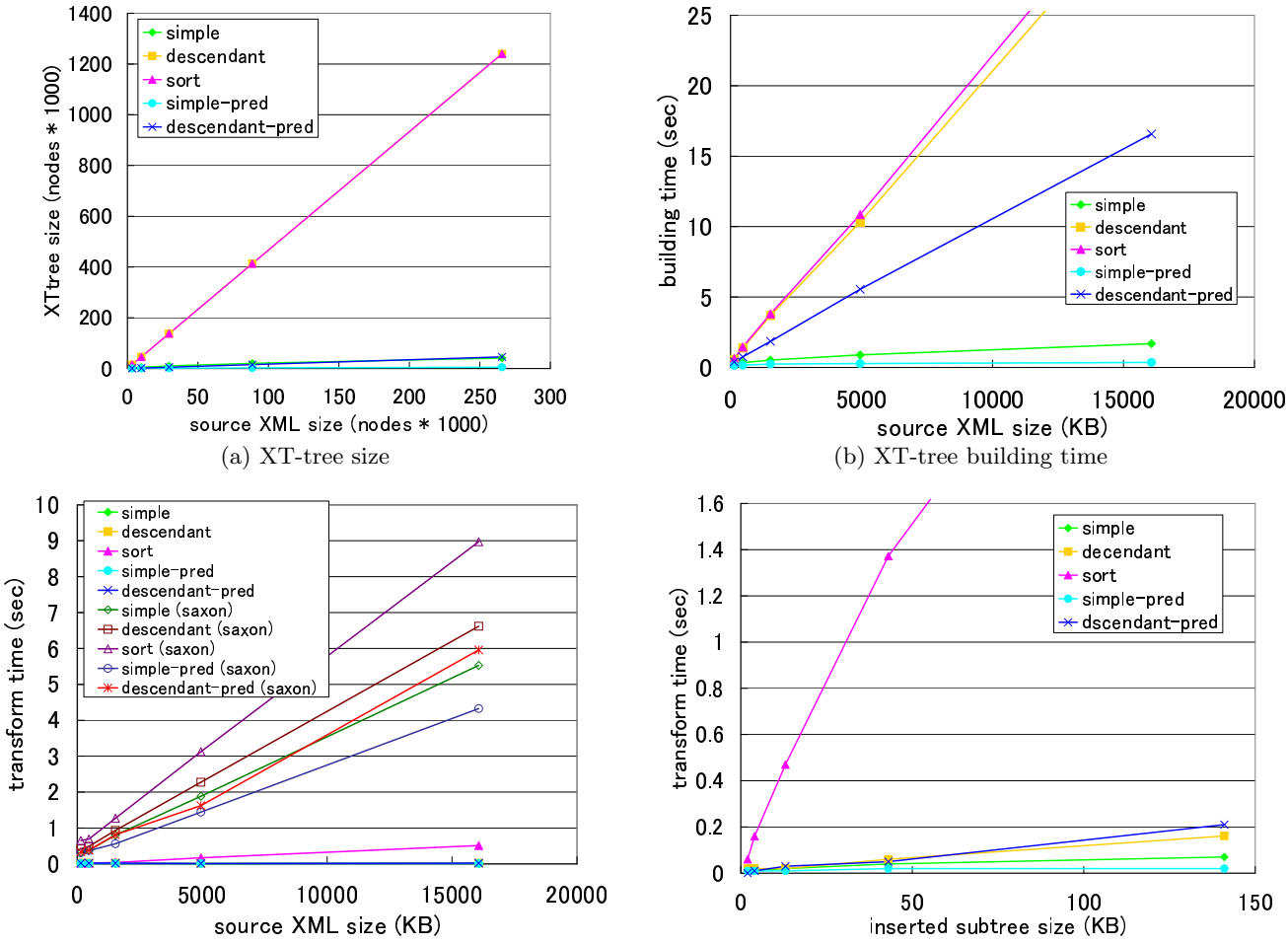
(c) in Fig. 14[6] shows the performance comparison of *XTim* and SAXON version 8.1 [15] (one of the fastest XSLT processors). The experiment compared the response time of 1) the *XTim* incremental transformation of subtree insertion with fixed size of 4KB, and 2) the SAXON transformation of the updated XML data.

There are three observations. First *XTim* outperforms SAXON's full transformation by factors of up to 500, because the updated data size is small compared to the source XML data size (2.84% in D7 to 0.02% in D11), so the XPath re-evaluation of *XTim* is localized. Second *XTim* performance of XSLT programs except **sort** is constant. The reason is that the insertion position of the new subtree is always the last position of the XT-node-set sorted by nodeID, because the PDOM assigns larger nodeID to a new node. In addition, *XTim* performance of **sort** is linear to $log_2(source\ XML\ data\ size)$, because a binary search is done on the XT-node-set to identify the insert position. Third, by comparing the XT-tree building time in (b) and the SAXON's transformation time in (c), the XT-tree building in **descendant**, **descendant-pred**, **sort** is slower than SAXON's transformation, but not in the **simple** and **simple-pred** cases. The reason for the former is that the XT-tree building process includes the XSLT transformation process. For the latter, we conjecture that the xsl:template lookup, which is one of the most expensive processes in XSLT transformation, is efficiently implemented in the *XTim* implementation due to the simplified XSLT specification described in Section 3.2.

6.3 Effect of updated data size

(d) in Fig. 14 shows the response times of *XTim* with the subtree insertion of various sizes. The source XML data is fixed at D10. The first observation is the large difference between **sort** and **descendant** performance; it indicates that the binary search cost incurred by every new XT-node insertion dominates **sort** performance. The second observation is **descendant-pred**'s performance worsens against that of

[6] *XTim* results except **sort** overlap.



(a) XT-tree size

(b) XT-tree building time

(c) transform time (4KB subtree insertion)

(d) transform time (D10 (4949KB) is the source XML)

Figure 14: Scalability experiments

descendant's as subtree size increases. This reflects the impact of the scope size difference on source XML data during XPath re-evaluation; the scope size of **descendant-pred** is larger than that of **descendant**.

7. RELATED WORK

There are a number of related works on incremental maintenance for materialized views in the context of the relational model [11], the semi-structured data model [17, 2, 21, 26], and the XML data model [25, 14, 9, 8].

incXSLT [25] is an incremental XSLT transformation algorithm for an XML document editor through its rendered presentations. *incXSLT* materializes the dynamic *execution flow* of XSLT template processing and traverses the flow in a top down manner. Their important contribution is to clarify how to manage XSLT template precedence for incremental view maintenance. XT-tree shares the concept of *execution flow*, but *XTim* offers three technical differences since their motivation differs from ours. First, although *incXSLT* incrementally maintains the *execution flow* of XSLT template processing, it doesn't incrementally maintain the XPath evaluation result and so re-evaluates XPath expressions fully. Second, *incXSLT* restricts update operations to just a single node, since they assume the update operation is done via GUI. This does not efficiently support the insertion/deletion of subtrees. Third, *incXSLT* requires that the XPEs whose result type is a node-set must be expressed by just child axis (not descendant axis). Reference [14] presents a maintenance technique for materialized XML views stored in a RDBMS or an ORDBMS. Their view definition is limited to select-projection views and it doesn't support regular expressions (wildcard and descendant axis) on matching patterns.

Reference [9, 8] presents an incremental XQuery view maintenance algorithm based on the algebra XAT. It constructs a materialized tree using XAT and checks matching XPEs with the update operation in a top down manner. There are three issues with their techniques. First, they don't consider predicates on XPath. Second, since they store all the intermediate node-sets returned by location step evaluations, they need a large memory space. Third, similar to *incXSLT*, they restrict update operations to just a single node that doesn't require descendant processing such as the **eval** algorithm in Fig. 7.

Reference [17] modifies the XML query language XML-QL to ensure multi-linearity. However, it doesn't support recursive matching patterns expressed by descendant axis in XPath and XML-QL is limited since it cannot express recursive queries expressed by an XSLT template. Reference [2] extends Lorel [3], a query language of the semi-structured

data model, and presents an incremental maintenance algorithm for materialized views defined by the extended Lorel. It records RelevantOids that contain the object identifier of every object touched during the view evaluation and checks whether RelevantOids contain the updated object. The views defined by the extended Lorel are limited to a subset of the source data, so it is not applicable to the views defined by XSLT programs which define more general data transformations. In addition, it restricts insertion/deletion to just the edges and value updates on atomic objects; match patterns are limited to those equivalent to $XP^{\{[]\}}$. References [21, 26] present incremental maintenance algorithms on the semi-structured data model whose views are limited to return a set of nodes.

8. CONCLUSION

We have presented *XTim*, a novel algorithm for incrementally maintaining XPath/XSLT views defined with XPath expressions in $XP^{\{[],*,//,vars\}}$. We investigated the XPath and XSLT features for incremental view maintenance in response to subtree insertion/deletion. *XTim* implements those features and experiments show that it improves the XML transformation performance by factors of up to 500.

There are several future research directions. *XTim* can be improved further with regard to restructuring transformations by considering some query containment. For example in Section 2, if an inserted subtree triggers the application of the second template for a certain year, we need to re-evaluate the XPath expression in line 15 only on the inserted subtree. The reason is the XPath expression collects publications for that year and no existence of the materialized result of the year indicates there is no publication on the year in existing source data. Other future work includes the efficient incremental maintenance of a large number of views and on an incremental update of the rendered presentations of HTML and SVG. We are currently working on the incremental SVG rendering to complete the incremental process from source data to presentation.

9. REFERENCES

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proceedings of SODA*, pages 547–556, 2001.

[2] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proceedings of VLDB*, pages 38–49, 1998.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[4] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.

[5] J. Clark. XSL transformations (XSLT) version 1.0, 16 November 1999. http://www.w3.org/TR/xslt.

[6] J. Clark and S. DeRose. XML path language (XPath) version 1.0, 16 November 1999. http://www.w3.org/TR/xpath.

[7] Y. Diao and M. J. Franklin. High-performance XML filtering: An overview of YFilter. In *IEEE Data Engineering Bulletin*, 2003.

[8] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. Order-sensitive view maintenance of materialized XQuery views. In *Computer Science Technical Report Series*. Worcester Polytechnic Institute, 2003.

[9] M. El-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner. An algebraic approach for incremental maintenance of materialized XQuery views. In *Computer Science Technical Report Series*. Worcester Polytechnic Institute, 2003.

[10] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proceedings of ICDT*, pages 173–189, 2003.

[11] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of SIGMOD*, pages 157–166, 1993.

[13] I. JTC1/SC29/WG11. Mpeg-7 description definition language (DDL). http://archive.dstc.edu.au/mpeg7-ddl/.

[14] H. Kang and J. Lim. Deferred incremental refresh of XML materialized views. In *Proceedings of Australasian database conference on Database technologies*, pages 217–226, 2003.

[15] M. H. Kay. SAXON the XSLT and XQuery processor. http://saxon.sourceforge.net/.

[16] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of VLDB*, pages 361–370, 2001.

[17] H. Liefke and S. B. Davidson. View maintenance for hierarchical semistructured data. In *Data Warehousing and Knowledge Discovery*, pages 114–125, 2000.

[18] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proceedings of Workshop on XML Data Management (XMLDM)*, LNCS. Springer, 2002.

[19] M. Onizuka, T. Honishi, F. Y. Chan, and R. Michigami. Incremental maintenance for materialized XPath/XSLT views (full version). http://xmltk.sourceforge.net/xtim.pdf.

[20] C. Owens. PMC project P/META. http://www.ebu.ch/departments/technical/pmc/pmc_meta.html.

[21] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *Proceedings of VLDB*, pages 227–238, 1996.

[22] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proceedings of SIGMOD*, pages 413–424, 2001.

[23] The XML:DB Initiative. XUpdate - XML update language, September 14 2000. http://www.xmldb.org/xupdate/.

[24] TV-Anytime Forum. http://www.tv-anytime.org/.

[25] L. Villard and N. Layaida. An incremental XSLT transformation processor for XML document manipulation. In *Proceedings of WWW*, pages 474–485, 2002.

[26] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *Proceedings of ICDE*, pages 116–125, 1998.