WReX: A Scalable Middleware Architecture to Enable XML Caching for Web-Services

Junichi Tatemura[1], Oliver Po[1], Arsany Sawires[2] *
Divyakant Agrawal[1], and K. Selçuk Candan[1]

[1] NEC Laboratories America
10080 North Wolfe Road, Suite SW3-350, Cupertino, CA 95014
{tatemura,oliver,agrawal,candan}@sv.nec-labs.com

[2] Department of Computer Science
University of California Santa Barbara
Santa Barbara, CA 93106
arsany@cs.ucsb.edu

Abstract. Web service caching, i.e., caching the responses of XML web service requests, is needed for designing scalable web service architectures. Such caching of dynamic content requires maintaining the caches appropriately to reflect dynamic updates to the back-end data source. In the database, especially relational, context, extensive research has addressed the problem of incremental view maintenance. However, only a few attempts have been made to address the cache maintenance problem for XML web service messages. We propose a middleware solution that bridges the gap between the cached web service responses and the back-end dynamic data source. We assume, for generality, that the back-end source has a general XML logical data model. Since the RDBMS technology is widely used for storing and querying XML data, we show how our solution can be implemented when the XML data source is implemented on top of an RDBMS. Such implementation exploits the well-known maturity of the RDBMS technology. The middleware solution described in this paper has the following features that distinguish it from the existing technology in this area: (1) It provides declarative description of Web Services based on rich and standards-based view specification language (XQuery/XPath); (2) No knowledge of the source XML schema is assumed, instead the source can be any general well-formed XML data; (3) The solution can be easily deployed on RDBMS, and (4) The size of the auxiliary data needed for the cache maintenance does not depend on the source data size, therefore, the solution is highly scalable. Experimental evaluation is conducted to assess the performance benefits of the proposed approach.

Keywords: web services, caching, XML views, path expressions, XML-relational mapping

* This work has been done during the author's summer internship at NEC

1 Introduction

Performance degradation of a Web Service can significantly impact the response times of front-end applications that use it. Especially for Web Services that provide dynamic content to many users (such as product information services), latency observed by the users is caused not only by the network transmission, but mainly by server overload at the back-end application. Offloading processing from the back-end applications is thus essential in providing Web Services scalability. Therefore, caching is a key enabling technology for scalable Web Service delivery.

A Web Service cache must handle request and response messages (typically formatted using XML); thus the cache must process (e.g., parse XML content of) a request message to identify the response message to be returned. Therefore, a standard HTTP cache cannot be directly employed when caching Web Services. Furthermore, in order to achieve loose coupling of remote services, Web Services usually handle messages with coarser granularities than traditional distributed object messaging such as CORBA. This fact makes it more difficult to map data source updates to the cached messages. Caching messages for data-driven Web Services thus requires middleware support for appropriate propagation of updates from the source to the cache.

It is commonly understood that an XML data/query model can be implemented on a relational model to leverage from the proven and highly-optimized storage and query capabilities already provided by existing relational database systems [15]. Thus, one approach to caching Web Service could be to apply existing technologies that manage data dependency between web content and data in relational databases, such as Data Update Propagation (DUP)[3], view invalidation [2], invalidation based on query templates [4], and many other works on view maintenance. However, these relational approaches will be very inefficient because an XML query can involve too many join operations when translated into SQL.

In this paper, we propose a middleware architecture, WReX, that bridges the semantic gaps among Web Service messages, a relational data model, and an XML data model, for caching Web Services. To make the proposed middleware solution applicable to various data sources, the WReX represents the source data in the caches as XML views and provides a declarative way to define Web Services to access the data. The WReX architecture (Sections 3 and 4) aims at resolving the impedance mismatch between the cached data content and the underlying database technology by applying recent XML-specific view maintenance techniques transparently in a relational setting.

Consequently, the WReX introduced in this paper consists of two complementary components: (1) Web Service Content Description (WSCD) mechanism fills the gap between Web Service messages and XML views of the source data and (2) XML view maintenance mapped to relational storage fills the gap between XML views and updates to the source data. This novel middleware architecture has the following features that distinguish it from the previous works: (1) It provides declarative description of Web Services based on rich and standards-based view

specification language (XQuery/XPath); (2) No knowledge of the source XML schema is assumed, instead the source can be any general well-formed XML data; (3) The solution can be easily deployed on RDBMS, and (4) The size of the auxiliary data needed for the cache maintenance does not depend on the source data size, therefore, the solution is highly scalable. Experimental evaluation is conducted to assess the performance benefits of the proposed approach. Experimental evaluations presented in Section 5 establish the performance benefits of the WReX middleware approach.

2 Cache-enabled Service Middleware Architecture

Figure 1 illustrates WReX, a Web Service middleware architecture enhanced with *web service caching*. WReX consists of a Web Service Application Server, an XML Data Source, and an Update Manager, which are implemented on top of a common Web computing platform (e.g., a J2EE application server and a relational database server). WReX lets users describe and deploy Web Services that deliver content generated from their own data sources. Given the description of a Web Service, the middleware manages request/response message caches.

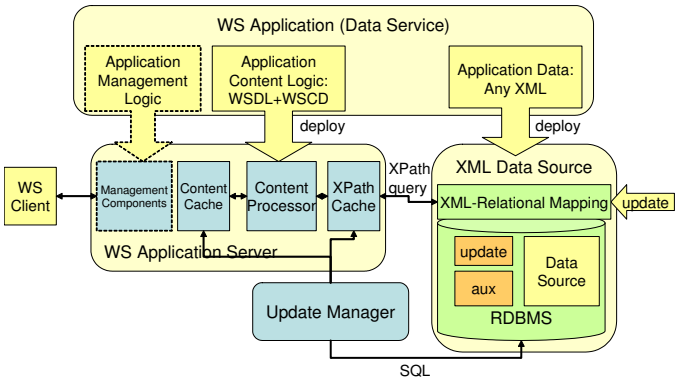


Fig. 1. WReX: Web Service Caching Architecture

A Web Service application is deployed on top of the WS Application Server and the XML Data Source as can be seen Figure 1. The application has three major parts: (1) data (data source to be published), (2) content logic (description of message content to be generated from the data source) , and (3) management logic (user authentication, logging, and metering). The *cache-enabled* Web Service application server consists of the following components: (1) Various management components, (2) a message content cache component, (3) a content processor, and (4) an XPath cache. Management components manipulate messages (e.g., insert data in the header) genereted by the content processor.

Management components handle management tasks such as user accounting and monitoring with approprite transformation of message content. Web service messages that contain management information are much less reusable even if actual content delivered to the user (e.g., product information) is reusable. By separating management functions as these components, WReX lets the other components focus on managing relationships between message content and the source data and makes cache more applicable.

The content logic specifies how to generate content of a message in response to a request message from a Web Service client. A shortcoming of the existing technologies is that, the Web Service definition language (WSDL) only defines interfaces (such as data types) of request/response messages, but does not provide content relationship between request and response messages [18]. To bridge this gap, we introduce a description platform, Web Service Content Description (WSCD), which provides a template of a response message that can contain references to data in a request message and queries to the source data. When the application server receives a request message, it generates a response message by integrating a message template and content fragments retrieved from the data source. Caching is applied to both generated response messages (Content Cache) and retrieved content from the source (XPath Cache).

This approach is similar to JSP (Java Server Pages) or ESI (Edge Side Includes). JSP provides a template of dynamic web pages and lets the application server construct a page from the template and content fragments generated by applications. Several application servers provide caching functionality for such content fragments in order to reduce application overload. ESI is a markup language used to define web content components for dynamic assembly and delivery of web pages at edge servers. The edge server dynamically integrates fragments into a web page and needs to retrieve only non-cacheable or expired fragments from the original servers. Datta et al. [5] has extended this approach to enable more flexible content composition on the edge server resulting in enhanced cacheability and reusability of content. In this sense, our approach can be seen as an extension of the JSP/ESI concept from HTML to XML context with XML cache update management. Another related example is the Weave management system [19] that enables the user to create Web content using declarative specification and caches various intermediary data such as views of relational data, XML page fragments, and HTML pages. Although it supports XML content generation from relational databases, update maintenance between cached XML content and data source is based on time stamps and specified with event-condition-action rules.

To enable caching of XPath queries to the data source as well as the message responses from the Web Service itself, the Update Manager needs to monitor updates in the data source and identify changes in the cached results. Here, note that an XML-aware data source is commonly implemented on an XML-aware RDBMS, which can leverage from the maturity of RDBMS implementations, extensive tuning, proven scalability, sophisticated query processing and query optimizers. However, even though the underlying DBMS is relational, tradi-

tional view/cache management solutions for relational data can not be directly applied to an XML data/query model. For example, CachePortal [2] automates cache update management based on a view invalidation technique in a relational model. However, when a query involves many join operations, which is the case of XML queries in a relational model, it is very inefficient due to costs from an extra database snapshot and over invalidation. Therefore, we introduce an update management middleware component which benefits from the relational nature of the back-end database, while deploying XML-specific view management techniques (i.e., the Update Manager that accesses the data source through SQL queries (Figure 1)).

2.1 Web Service Content Description (WSCD)

Given a service request, the Web Service generates response messages based on the service logic. The interface between the request and response is usually defined using WSDL (Web Service Definition Language). WSDL, on the other hand, does not describe content relationships between request and response messages, which are needed for managing updates. We propose Web Service Content Description (WSCD) language that describes how a response message is generated for a given operation specified in WSDL. Formally, the WSCD for a service operation o consists of three parts: (V, T, S), where V is the variable assignment definition, T is the template definition, and S is the source references.

- The variable assignment definition V defines how to extract data from a request message. Mapping from a request message to variables is given by pairs of name and XPaths: $V = \{(name_i, xpath_i)\}$. Given a request message, which can be seen as an XML document, V generates a specific variable assignment $v = \{name_i = value_i\}$. In addition to the generation of a response message, v is used as the identity of the message cache: the identity consists of an operation name and a variable assignment (o, v).
- The template T defines the content of a response message with references to the variables V. The template can contain XQuery expressions to dynamically insert data derived from the data source.
- The source reference S maps URIs of data source service endpoints to document URIs referred to by XQuery expressions in T.

Figure 2 shows an example of a WSCD description. Elements `<cd:Variables>`, `<cd:Template>`, `<cd:ServiceEPR>` correspond to (V, T, S), respectively.

A variable is defined with a part of the request message (i.e. input) of a WSDL operation and an XPath expression that indicates data within the part. Combined with WSDL binding information, it is translated to a full XPath expression applied to a request message, for example:

"`/Envelope/Body/GetBookRequest/Category/text()`"

in case of the SOAP literal binding. A template specifies an XML content of a part of the response message (i.e., output) of a WSDL operation. It can contain

```
<cd:WSCD xmlns:cd=... operation="GetBook">
 <cd:Variables>
  <cd:Let name="category" part="body"
     path="/GetBookRequest/Category/text()"/>
  <cd:Let name="maxprice" part="body"
     path="/GetBookRequest/Max/text()"/>
  <cd:Let name="minprice" part="body"
     path="/GetBookRequest/Min/text()"/>
 </cd:Variables>
 <cd:Template part="body">
  <GetBookResponse>
    <cd:Query>FOR ... LET... WHERE... RETURN...</cd:Query>
  </GetBookResponse>
 </cd:Template>
 <cd:ServiceEPR .../>
</cd:WSCD>
```

Fig. 2. Example of Web Service Content Description

an XQuery specified in `<cd:Query>`. The query may refer to variables defined in the variables part.

Note that WSCD is meant to provide a simple specification of message content in a request-response Web Service operation. If the user wants a full set of programming functionality to create Web Service (such as event handling), a special programming language for Web Services, such as XL [8], could be used instead of WSCD. In fact, since XL uses XQuery expressions to access data, a possible extension of WReX is to support the XL language, in addition to WSCD, for services with complicated interactions.

Our WSCD approach is also related to "declarative web services" [1], used for composing dynamic XML documents by importing fragments. For optimized data management, a declarative web service that provides fragments is defined as an XQuery on data sources. Although they focus on data replication issues in a distributed environment, they also state possibility of querying cost reduction through an update propagation mechanism, on which we focus in this paper.

2.2 Cache Management using WSCD

The WSCD description of Web Service messages provides a framework to manage Web Service caching. First, the system needs to identify the matching incoming requests and cached response messages. This task is done by extracting values from an incoming message with XPath expressions in the variable definition V since the cache identity is given as a variable assignment (o, v). Efficient filtering [7] can be applied to process multiple XPath matching results in a scalable manner. Then we focus on the second task: to manage update dependencies between cached messages and the data at the source.

As described above, the WSCD template contains a set of XQuery expressions $XQ = \{xq_i\}$ to insert dynamic data from the source into response messages. Since an XQuery expression xq contains references to the variables V and the source S, what the system needs to manage is an XQuery instance (xq, v, S): when the result of an XQuery instance is updated, the message cache items that contain this result must be updated or invalidated.

An XQuery statement accesses documents (i.e., the source data) through XPath expressions. Thus, a set of XPath expressions $XP = \{xp_i\}$ is extracted from XQueries XQ and is given to the XPath cache component, which caches an XPath instance: (xp, v, S). The XPath Cache receives an XPath query from XQuery Processor and returns the query result from the cache. If it is not cached, the XPath Cache issues an XPath query to the data source. The data source returns the query result and makes available *auxiliary data* required to maintain XPath cache (Section 3).

When the Update Manager observes updates in the data source, it determines the impact of the source update to cached XPath results. During this process, the Update Manager uses the auxiliary data and update data to identify the cache updates. It may also access the source data if needed. Then it maintains cached results in the XPath Cache affected by the update. Consequently, message cache items that refer to the affected XPath instances are also either invalidated or maintained. In order to effectively manage update dependency between message cache and the data source, the WReX uses our XML-specific view maintenance techniques described next.

3 XPath Cache Maintenance

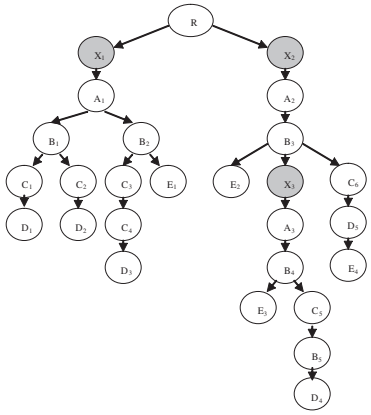
In this section we describe the data model and the incremental XPath maintenance technique WReX relies on. Further details of both are presented in [13].

3.1 Data Model

As described earlier, the underlying logical model of the data source is XML. Each XML data source is represented as an ordered tree in which every node n is a pair $\langle n.id, n.label \rangle$ where $n.id$ is a node identifier that uniquely identifies the node and $n.label$ is a string that describes the node type and/or value. We use upper-case letters to represent the node labels. For example, A, B, and C are node labels. We use numeric subscripts to distinguish different nodes that have the same label. Thus, A_i and A_j refer to two distinct nodes with the same label A. Figure 3 shows an example document tree and path expression that will be used as a running example to illustrate the incremental maintenance technique.

3.2 Update Model

A source update is a transformation of the source XML document. Any source transformation can be expressed in terms of the two primitive operations of



(a) XML Data

$$/\mathbf{A}//\mathbf{B}[\mathbf{Count}(//\mathbf{E}) \geq \mathbf{1} \vee \mathbf{Count}(/\mathbf{D}) \geq \mathbf{1}]//\mathbf{C}[\mathbf{Count}(//\mathbf{E}) = \mathbf{0}]//\mathbf{D}$$

(b) XPath Query

Fig. 3. (a) An Example XML Tree and (b) a path-expression $\mathcal{E}$

addition and deletion of leaf nodes. Thus, for simplicity, in this section, we focus on the maintenance operations needed to handle these two types of source updates. Formally, we model a source update $\mathcal{U}$ as a pair $\langle \mathcal{U}.type, \mathcal{U}.path \rangle$ where $\mathcal{U}.type$ is the type of the update: *Add* (add a leaf node) or *Delete* (delete a leaf node). $\mathcal{U}.path$ is the path of all the ancestors of the added or deleted node starting with the document root and ending with the added or deleted node itself. The added or deleted node itself is referred to as $\mathcal{U}.node$. For example, $\mathcal{U} = \langle Add, (R, X_1, A_1, B_1, Z) \rangle$ represents the addition of node Z as a child node of node B_1 in the XML document shown in Figure 3(a).

3.3 Query Model

Path expressions are the basic building blocks of XML queries and therefore are fundamental to implementing Web Services in our framework. The cache content is the result of applying path expression-based queries to the source document. A path expression $\mathcal{E}$ of size N is a sequence of N steps: $(s_1, s_2, \cdots s_N)$. A step s_i is a triple $\langle s_i.axis, s_i.label, s_i.pred \rangle$ where (i) $s_i.axis$ is an axis test (child '/' or descendent '//'); (ii) $s_i.label$ is a label test; and (iii) $s_i.pred$ is an optional predicate test which can be any complex condition examining the labels and the structure of the nodes in the subtree of the node being tested. $Pred_i(n)$ is said to be *true* if and only if (1) Node n belongs to the source tree, and (2) $s_i.pred$ evaluates to *true* at node n or step s_i does not have a predicate test. For example, $Pred_3(C_1)$ in the example is *true* because C_1 satisfies the condition $s_3.pred$ since C_1 has no descendants labeled E.

Given an expression $\mathcal{E}$, a document tree $\mathcal{D}$, and a sequence of context nodes $\mathcal{C}$ (the set of staring nodes from $\mathcal{D}$), a query, $\mathcal{Q} = q(\mathcal{E}, \mathcal{C}, \mathcal{D})$ returns a sequence of nodes $\mathcal{R}$ as a result. For example, consider the query $\mathcal{Q} = q(\mathcal{E}, \mathcal{C}, \mathcal{D})$ where: $\mathcal{D}$ is the document tree shown in Figure 3(a), $\mathcal{C} = (X_1, X_2, X_3)$ are the shaded nodes the same figure, and $\mathcal{E}$ is the path expression specified in Figure 3(b). Given this query,

1. the first step s_1 $(/A)$ starts at every node in $\mathcal{C}$ and selects all the children with label A; this results in the first intermediate result $\mathcal{R}_1 = (A_1, A_2, A_3)$.
2. s_2 $(//B[Count(//E) \geq 1 \vee Count(/D) \geq 1])$ starts at every node in $\mathcal{R}_1$ and selects all the descendants with label B that have at least one descendant labeled E or at least one child labeled D; this results in the second intermediate result $\mathcal{R}_2 = (B_2, B_3, B_4, B_4, B_5, B_5)$. Note that B_4 - and also B_5 - occurs twice in $\mathcal{R}_2$ because it can be derived in two ways from nodes of $\mathcal{R}_1$, one from A_2 and another one from A_3.
3. starting at $\mathcal{R}_2$, step s_3 $(/C[Count(//E) = 0])$ selects all the descendants labeled C that have no descendants labeled E; this results in $\mathcal{R}_3 = (C_3, C_4, C_5, C_5, C_5)$.
4. finally, s_4 $(//D)$ starts at $\mathcal{R}_3$ and selects all the descendants labeled D. Hence, the final result of $\mathcal{Q}$ is $\mathcal{R} = \mathcal{R}_4 = (D_3, D_3, D_4, D_4, D_4)$.

We differentiate between the multiple occurrences of the same node in a result by using a numeric superscript. For example, we denote the result $\mathcal{R}$ as $\mathcal{R} = (D_3^1, D_3^2, D_4^1, D_4^2, D_4^3)$.

For a node $n \in \mathcal{R}$, the sub-sequence of the ancestors of a node n that matched the steps of $\mathcal{E}$, and thus caused n to appear in $\mathcal{R}$ is referred to as the *result path* of n and denoted as $ResultPath(n)$. $ResultPath_i(n)$, where $i \geq 0$, is the i^{th} element in $ResultPath(n)$. In the example query above, $ResultPath(D_3^1) = (X_1, A_1, B_2, C_3, D_3)$ and $ResultPath(D_3^1)_2 = (X_1, A_1, B_2, C_3, D_3)$ is B_2.

3.4 Incremental Maintenance of Path Expression Results

A source update $\mathcal{U}$ can affect the cached result $\mathcal{R}$ by adding or deleting nodes to any of the intermediate results $\mathcal{R}_i$. The primary reason of such additions and deletions is changing the truth values of the expression predicates at the steps of the expression:

> If an update changes a predicate $Pred_i(n)$ from $false(true)$ to $true(false)$, we say that the update directly adds (deletes) node n at step i.

A direct addition (deletion) at step i can induce other indirect additions (deletions) in steps $j > i$. The final result $\mathcal{R}$ is affected if and only if the effect propagates all the way to step N. For example, if $\mathcal{U} = (Add, (R, X_1, A_1, B_1, E_5))$, then $Pred_2(B_1)$ changes from *false* to *true*. The direct effect of this is to add B_1 to $\mathcal{R}_2$. The resulting indirect effects are the addition of C_1 and C_2 to $\mathcal{R}_3$ and then the addition of D_1 and D_2 to $\mathcal{R}_4$. For each step, the incremental maintenance process first discovers all the direct effects and then uses these effects to discover the indirect ones.

Discovering the Direct Effects of the Updates. We identify the direct effects of the updates in two phases: **Axis&Label test** and the **predicate test**.

Phase I - Axis&label test: Let us define δ_i^+ and δ_i^- as the sequences of all nodes that $\mathcal{U}$ directly adds/deletes at $\mathcal{R}_i$ respectively. Let also $\delta_i = \delta_i^+ \sqcup \delta_i^-$. The job of this phase is to identify a sequence Δ_i such that we can guarantee, without any source queries, that $\delta_i \sqsubset \Delta_i$.

In [13], we showed that every node n in δ_i must also belong to $\mathcal{U}.path$. Moreover, for a node n to be directly added to be in δ_i, it must have an ancestor in every $\mathcal{R}_j$, $j < i$. Since n itself belongs to $\mathcal{U}.path$, then all its ancestors also belong to $\mathcal{U}.path$. This suggests that $\mathcal{U}.path$ has much of the information needed to identify the nodes of δ_i. In fact, applying the axes and labels tests to $\mathcal{U}.path$, ignoring the predicate tests, provides a sequence Δ_i which is guaranteed to be a supersequence of δ_i. This is because this process uses a relaxed selection condition (it ignores the predicate tests, which evaluation requires querying the source) over the branch $\mathcal{U}.path$ which is guaranteed to include all the nodes of all the δ_i's. Computing the Δ_i's from $\mathcal{U}.path$ proceeds very similar to computing the $\mathcal{R}_i$'s from the source tree $\mathcal{D}$. For example, consider an update $\mathcal{U}$ of adding a node D_6 as a child of D_4. In this case, $\mathcal{U}.path$ is the tree branch that starts with the root R and ends with D_6. Computing the different Δ_i's as described above results in: $\Delta_0 = (X_2, X_3)$, $\Delta_1 = (A_2, A_3)$, $\Delta_2 = (B_3, B_4, , B_4, B_5, B_5)$, $\Delta_3 = (C_5, C_5, C_5)$, $\Delta_4 = (D_4, D_4, D_4, D_6, D_6, D_6)$. Note that the only nodes that will be directly added are the three occurrences of D_6 that appear in Δ_4; all the other nodes n in all the computed Δ_i's will not be added or deleted because $\mathcal{U}$ did not affect $Pred_i(n)$. Note that, because D_6 did not exist before $\mathcal{U}$ occurred, the value $Pred_i(D_6)$, $\forall i$ is *false* before $\mathcal{U}$. Similarly, if an update deletes a node n from the source tree, the value $Pred_i(n)$, $\forall i$ is *false* after $\mathcal{U}$.

Phase II - Predicate test: This phase identifies the exact sequence δ_i by determining which nodes in Δ_i had their predicate values changed due to the update.

To detect such changes we need to compare, for every node in δ_i, the values of $Pred_i(n)$ before and after $\mathcal{U}$ occurred. Let us denote the value of the predicate before the update occurred as $Pred_i^{before}(n)$ and the value after the update as $Pred_i^{after}(n)$. The value of $Pred_i^{after}(n)$ can be easily calculated by querying the source. The value of $Pred_i^{before}(n)$, on the other hand, cannot be computed by a source query because the update $\mathcal{U}$ has already been incorporated at the source. Once again, in [13], we showed that we can deduce the value of $Pred_i^{before}(n)$ using the information of the result paths. Specifically, we showed that if we define $RP_i(n)$ to be *true* if and only if n is the i^{th} element of the result path of some node in $\mathcal{R}$, then we can take $Pred_i^{before}(n) = RP_i(n)$. Therefore, we keep the result paths' information as auxiliary data with the cached result $\mathcal{R}$. With that, we compute $Pred_i^{before}(n)$ without issuing any source queries. To compute the size of this auxiliary data, recall that each result path is of length $N + 1$; if M is the size of the cached result $\mathcal{R}$, then the size of the auxiliary data is clearly

Incremental_Maintenance (Expression $\mathcal{E}$, Update $\mathcal{U}$)

1- $\Delta_0 = \mathcal{C} \cap \mathcal{U}.path$
$\mathcal{R}^+ = \mathcal{R}^- = ()$ //Empty sequences
$i = 1$ // loop variable

2- WHILE ($i \leq N$ AND Δ_{i-1} is not empty)

2-1 $j = i$
WHILE (s_j has no predicate test AND $j < N$) j++

2-2 $\Delta_j = q((s_i, s_{i+1}, \cdots, s_j).axis\&label, \Delta_{i-1}, \mathcal{U}.path)$

2-3 Let $\mathcal{T}_j = (n | n \in \Delta_j \wedge Pred_j^{after}(n) = true)$

2-4 $\delta_j^+ = (n | n \in \mathcal{T}_j \wedge RP_j(n) = false)$

2-5 $\mathcal{R}^+ = \mathcal{R}^+ \sqcup q((s_{j+1}, s_{j+2}, \cdots, j_N), \delta_j^+, \mathcal{D})$

2-6 $\mathcal{R}^- = \mathcal{R}^- \sqcup (n | n \in \mathcal{R} \wedge ResultPath_j(n) \in (\Delta_j - \mathcal{T}_j))$

2-7 $\Delta_j = \mathcal{T}_j - \delta_j^+$

2-8 $i = j + 1$

3- $\mathcal{R} = \mathcal{R} \sqcup \mathcal{R}^+$
$\mathcal{R} = \mathcal{R} - \mathcal{R}^-$

Fig. 4. Incremental View Maintenance Algorithm for XML Path Expressions

$O(M * N)$. Thus the auxiliary data size is bounded by the expression size and the result size and it does not depend on the source data size.

Discovering the Indirect Effects of the Updates To discover the indirect effects from the direct ones, we need to handle two cases:

1. *Indirect additions due to direct additions:* when a node n is directly added to $\mathcal{R}_i$ then, in order to retrieve the indirect additions at $\mathcal{R}$, the maintenance algorithm issues a source query with context as n and with the steps sequence $(s_{i+1}, s_{i+2}, \cdots, s_N)$. This query is denoted as $q((s_{i+1}, s_{i+2}, \cdots, s_N), (n), \mathcal{D})$.
2. *Indirect deletions due to direct deletions:* when a node n is directly deleted from $\mathcal{R}_i$, then all the nodes $r \in \mathcal{R}$ that came to $\mathcal{R}$ due to n belonging to $\mathcal{R}_i$ must also be deleted from $\mathcal{R}$. These are the nodes $r \in \mathcal{R}$ which have $ResultPath_i(r) = n$. Thus, using the auxiliary data described above, we can discover the indirect deletions without issuing any source queries.

The Full Algorithm. Figure 4, shows an algorithm based on the ideas presented above. Step 1 initializes some algorithm variables. $\mathcal{R}^+$ and $\mathcal{R}^-$ are the sequences of nodes to be added and deleted, respectively, in $\mathcal{R}$. The loop in step 2 computes the different $\Delta's$. Step 2-1 assigns the value of j such that the range $i : j$ spans all the expression steps starting at i that do not have predicate tests. For this range, no predicate tests are needed because all the predicates are known to be *true*, by definition, before and after $\mathcal{U}$. Thus, there are no direct effects in this range. Therefore, the algorithm combines all the axis&label tests of this range in one step, namely, step 2-2. Step 2-3 identifies $\mathcal{T}_j$ as the sequence of the nodes of Δ_j that have $Pred_j^{after}(n) = true$. Step 2-4 then discovers the direct additions at $\mathcal{R}_j$. These direct additions are then used by step 2-5 to discover the indirect effects on $\mathcal{R}$. Step 2-6 discovers all the ultimate deletions at $\mathcal{R}$, it

implicitly discovers the direct deletions and uses them to discover the indirect ones. Step 2-7 excludes from Δ_j the nodes that will not have effects on later iterations, this is formally proved in [13]. Step 2-8 increments the loop variable to start after j in the next step. Finally, step 3 updates $\mathcal{R}$ using $\mathcal{R}^+$ and $\mathcal{R}^-$.

Note that the algorithm does not differentiate between source addition and deletion updates, the only case that needs to make such distinction is when $\mathcal{U}.node$ itself belong to Δ_N, this case is implicitly taken care of in the computation of $Pred_i(n)$ before and after $\mathcal{U}$

In addition to the result $\mathcal{R}$, the auxiliary data also need to be maintained. This is not shown here for simplicity.

In the following section, we show how this algorithm is implemented when the source XML document is stored in an RDBMS and hence, queried by SQL queries.

4 Implementation over RDBMS

Although there have been several efforts to build native XML database systems [10, 11], a common consensus is to use RDBMS technology to leverage from the proven and highly-optimized storage and query capabilities already provided by existing relational database systems [15].

Therefore, in this section, we show how the incremental XPath maintenance algorithm described in Section 3 can be implemented when RDBMS technology is used for the storage of the XML source data, the auxiliary data, and the cached results. This requires an update management middleware which bridges the gap between the XML logical data model at one side, and the relational database implementation at the other side.

First, we will describe the XML-to-RDBMS and XPath-to-SQL mapping schemes the middleware uses (Section 4.1). Then we will describe how to employ this relational framework for incremental view maintenance of XPath queries to support efficient Web Service caching (Section 4.2).

4.1 Storing and Querying XML over RDBMS

XML Data to Relational Data Mapping Given the mismatch between the XML data model (which has a nested structure) and the relational data model (which is flat), several techniques have been proposed for storing and querying XML documents using relational database systems [6, 9, 16, 15]. These approaches typically work as follows. The first step is *relational schema generation*, where relational tables are created for the purpose of storing XML documents. The next step is XML document *shredding*, where XML documents are stored by shredding them into rows of the tables that were created in the first step. The final step is XML query processing (XPath queries in our case), where XPath queries over the stored XML documents are converted into SQL queries over the created tables.

id	label	type	value	parent
1	Manuscripts	element	NULL	0
1.1	Category	attribute	Fiction	1
1.3	Book	element	NULL	1
1.3.1	ISBN	attribute	1-555860-438-3	1.3
1.3.3	Title	element	NULL	1.3
1.3.3.1	NULL	value	A Story	1.3.3
1.3.5	Author	element	NULL	1.3
1.3.5.1	Country	attribute	USA	1.3.5
1.3.5.3	NULL	value	John Doe	1.3.5
1.5	Monograph	element	NULL	1
1.5.1	ISBN	attribute	1-888570-843-5	1.5
1.5.3	Title	element	NULL	1.5
1.5.3.1	NULL	value	Another Story	1.5.3
1.5.5	Author	element	NULL	1.5
1.5.5.1	Country	attribute	Canada	1.5.5
1.5.5.3	NULL	value	Tom Alter	1.5.5

Fig. 5. SrcTBL: The XML Document Table

One simple approach of shredding is to store each node in the XML tree as a tuple in a relational table, which maintains all the necessary information, such as the node label, and node type. *Node identifiers* are used to capture and represent the structure of the XML source in the relational database. In order to efficiently maintain path-expression views over XML documents, two essential properties must be provided by node identifiers: First, element(s) updated in the source XML document should be easily identified. Secondly, structural (parent, child, descendent, sibling) relationships among the elements of the XML document should be easily determined using the node identifiers. These are critical for efficient query processing and also in facilitating effective view maintenance in the presence of updates.

Several approaches are proposed to assign node identifiers to the nodes in XML document. We apply one such approach called, the ORDPATH [12] scheme (also used in the upcoming version of Microsoft SQL Server). ORDPATH identifiers can be assigned to the nodes of an XML tree without requiring a schema. ORDPATHs are conceptually similar to the Dewey Order introduced in [17]. The resulting identifiers have the property that ancestor relationships between the nodes is captured by the prefix relationship between the corresponding node identifiers: $ancestor(n_i, n_j) \leftrightarrow prefix(n_i.nid, n_j.nid)$.

Consider the following sample XML document:

```
<Manuscripts Category="Fiction">
   <Book ISBN="1-555860-438-3">
       <Title>A Story</Title>
       <Author Country="USA">John Doe</Author>
   </Book>
   <Monograph ISBN="1-888570-843-5">
       <Title>Another Story</Title>
      <Author Country="Canada">Tom Alter</Author>
  </Monograph>
</Manuscripts>
```

Figure 5 shows the table SrcTBL in which an XML document is stored in an RDBMS

- `id`: The ORDPATH identifier originally proposed is implemented as a bit string, and an RDBMS is supposed to implement primitive functions for structural relationships and query plans optimized for ORDPATHs. In our prototype, we have implemented an ORDPATH id as a character string, as shown in Figure 5, for experimental purpose without implementing primitive functions in RDBMSs. The primitive $ancestor(n_i.id, n_j.id)$ is implemented as a string prefix matching: "$n_i.id$ `LIKE` $n_j.id$ `||` `'%'`". Note that the node id column captures the order of the XML document, thus this XML order semantics are not lost when the document is stored in an unordered relational system.
- `parent`: To identify a parent-child relationship effectively in our experimental prototype, we additionally store the parent node id in the table. The primitive $parent(n_i.id, n_j.id)$ is in fact implemented as "n_i`.id =` n_j`.parent`".
- `label, type, value`: A node type is specified in `type` , which is either an `element`, `attribute`, or `value`. An `element` node has its tag name in `label`. An `attribute` node has its name and value in `label` and `value` respectively. A `value` node has its value in `value`. Although our view maintenance algorithm is presented on a simplified document model (i.e., $\langle n.id, n.label \rangle$), it can be easily mapped in this node model.

With this table schema in place, XPath queries can be processed by translating them into SQL queries against a table of this schema, as illustrated next.

4.2 XML Document Update Management

For each cached XPath expression, the system stores the following data required for incremental maintenance (Section 3): (1) `CntxtTBL`: a table of the nodes that comprise the query context, (2) *Query Statement:* an SQL representation of the original XPath expression, (3) *Individual query step:* an SQL representation of each step in the incremental maintenance algorithm, and (4) `AuxTBL`: the auxiliary data (i.e. the result paths), whose schema is `AuxTBL(id0, id1, id2,` $\cdots$`,` `id`N) (where N is the number of steps in the cached expression, each row in this table stores a result path of the result, and the nodes in the last column `id`N comprise $\mathcal{R}$).

In the maintenance process, the whole auxiliary data (i.e., `AuxTBL`) needs to be maintained, not only the final result $\mathcal{R}$ which is stored in the last column of that table. We have implemented that simply by projecting more columns in the SELECT clauses of the following SQL statements. With that, the rows resulting from these SQL statements represent partial path expressions. Therefore, we use join operations to concatenate these partial result paths to form full result paths to maintain `AuxTBL`. For simplicity, we do not show the concatenation queries here.

In addition to these tables, we maintain an update table (`UpdtTBL`) that stores the source update being processed. As mentioned before, each update $\mathcal{U}$ is represented by $\mathcal{U}.path$ which is a branch of the source tree. Thus, we use the same schema as for the `SrcTBL`.

The View Maintenance Process We illustrate the view maintenance process with the folowing expression as an example:

$$/site/person[LIKE(@id,"person\%")]/name$$

To construct the SQL query representing this expression, the hierarchical relationships between the nodes can be represented by either nested SQL queries or as self-join operations on the source table, `SrcTBL`, shown in Figure 5. We adopted the second option in our solution because it allows the query optimizer to generate more efficient query plans. Thus, the expression is transformed into the following SQL query by the middleware:

```
SELECT A.id, B.id, C.id, E.id
FROM CntxTBL A, SrcTBL B, SrcTBL C, SrcTBL D, SrcTBL E
WHERE parent(B.id)=A.id AND parent(C.id)=B.id AND parent(D.id)=C.id
AND parent(E.id)=C.id
AND B.type = 'element' AND A.label = 'site'
AND C.type = 'element' AND B.label = 'person'
AND D.type = 'attribute' AND D.label = 'id' AND LIKE(D.value,'person%')
AND E.type = 'element' AND E.label = 'name'
```

In this query, the final result is the set of nodes in the last projection E.id, the other projections A.id, B.id and C.id represent the result path information which is used as auxiliary data for the maintenance process.

The algorithm in Figure 4 starts by initializing Δ_0 in step 1 by an intersection operation:

```
CREATE TABLE Δ0(id0) AS
(SELECT id FROM CntxtTBL INTERSECTION SELECT id FROM UpdtTBL)
```

Then, in the first iteration of the loop, step 2-1 assigns to j the value 2 because s_1 has no predicate test. Then, step 2-2 computes Δ_2 by the following SQL statement:

```
CREATE TABLE Δ2(id0, id1, id2) AS
SELECT A.id, B.id C.id FROM Δ0 A, UpdtTBL B, UpdtTBL C
WHERE parent(B.id)=A.id AND parent(C.id)=B.id
AND B.type = 'element' AND B.label = 'site'
AND C.type = 'element' AND C.label = 'person'
```

The projection of A.id and B.id here are to get partial result paths.
In step 2-3, $\mathcal{T}_2$ is computed by:

```
CREATE TABLE T2 AS SELECT A.id FROM Δ2 A, SrcTBL B
WHERE parent(B.id)=A.id
AND B.type = 'attribute' AND C.label = 'id'
AND LIKE(B.value,'person%')
```

Then step 2-4 computes the direct additions at $\mathcal{R}_2$ as follows:

```
CREATE TABLE δ2+ AS
SELECT T.id FROM T2 T
WHERE NOT EXISTS (SELECT * FROM AuxTBL WHERE id2 = T.id)
```

Step 2-5 then uses δ_2^+ to discover the ultimate additions at $\mathcal{R}$, the SQL query used to discover these additions is:

```
SELECT A.id, B.id FROM δ2+ A, SrcTBL B
WHERE parent(B.id)=A.id
AND B.type = 'element' AND B.label = 'name'
```

(A.id, B.id) in this query result is a partial result path starting at $\mathcal{R}_2$ until $\mathcal{R}_3$.

Then step 2-6 computes the ultimate deletions at $\mathcal{R}$ as follows:

```
SELECT DISTINCT A.id3 FROM AuxTBL A
WHERE A.id2 IN
SELECT id2 FROM Δ2 DIFFERENCE SELECT id FROM T2
```

step 2-7 simply reduces Δ_2 by a DIFFERENCE operator.

In the second (also, last) iteration of the loop, we have $i = j = 3$. In step 2-2, Δ_3 is computed from the reduced Δ_2. Since this iteration is processing the last expression step, then if $\mathcal{U}.node$ belongs to Δ_3 then the computation of $Pred_3(\mathcal{U}.node)$ takes into account $\mathcal{U}.type$. This is computed as follows: If $\mathcal{U}.type = Add$, then $Pred_3^{before}(\mathcal{U}.node) = false$ because $\mathcal{U}.node$ did not exist in the source before $\mathcal{U}.node$. If $\mathcal{U}.type = Del$, then $Pred_3^{after}(\mathcal{U}.node) = false$ because $\mathcal{U}.node$ does not exist in the source after $\mathcal{U}.node$. These two cases are implicitly taken care of in the algorithm without testing $\mathcal{U}.type$ in the computation of $Pred_3(\mathcal{U}.node)$ before and after $\mathcal{U}$. Finally, all the ultimate additions and deletions in AuxTBL are determined by joining the partial result paths discovered by the SQL queries shown above.

5 Experimental Evaluation

In this section, we experimentally show that the proposed scheme provides a large performance impact, while incurring a small storage and processing overhead. For this purpose, we used the XMARK benchmark [14] to generate a data set of 325,236 nodes. Experiments are done using an Oracle 9i database on a PC with Linux 8.0, Pentium 4 1800 MHz CPU with 1 GB memory. We evaluated the caching performance by using the following XPath queries:

- *XP1*: /site/people/person[like(@id,"person%")]/name/text()
- *XP2*: /site/closed_auctions/closed_auction[price>40]/price/text()
- *XP3*: /site//item[contains(description,"gold")/name/text()
- *XP4*: /site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/emph/ketword/text()

Overhead of Auxiliary Data Table 1 shows the overhead of auxiliary data (i.e., `AuxTBL`) in terms of storage requirements and execution time. In addition to cached XPath results (denoted as columns R-VAL and R-ID), the system

	R-VAL (byte)	R-ID (byte)	AUX (byte)	SOV	FQ (msec)	FQA (msec)	EOV
XP1	36538	30103	85199	1.28	532	551	1.04
XP2	2366	8312	24267	2.27	802	876	1.09
XP3	3080	2327	6096	1.13	3933	4019	1.02
XP4	964	752	5525	3.22	3520	3556	1.01

Table 1. Overhead in Auxiliary Data Maintenance: R-VAL: Result Set Value Storage, R-ID: Result Set Node ID Storage, AUX: Auxililary Data Storage, SOV: Storage Overhead (=AUX/(R-VAL+R-ID)), FQ: Full Source Query Execution Time, FQA: Full Source Query with Aux. Data Execution Time, EOV: Execution Time Overhead (=FQA/FQ).

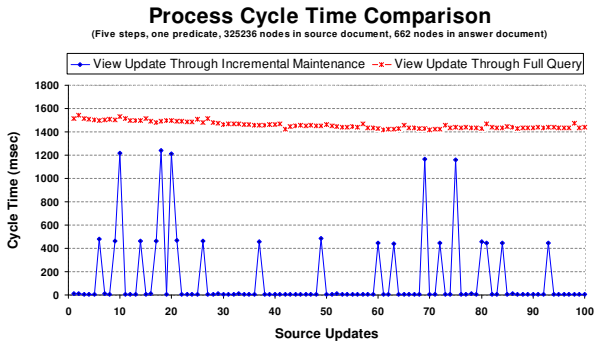
needs to store result paths as auxiliary data(AUX). As can be seen in the AUX column, the storage overhead does not depend on the data size, but depends on the number of steps in the XPath query and the cached data size. Then, to observe the query processing in WReX, we compared the original full query execution time with the execution time of the modified query that also retrieves result paths to be used as auxiliary data. As shown in the Table 1, the overhead is less than 10% in each case.

Performance Impact of Cache-enabled Middleware To observe the benefit of WReX in reducing the execution time observed by the users, we have compared the execution time requirements for incremental cache update and full recomputation on the following cached queries:

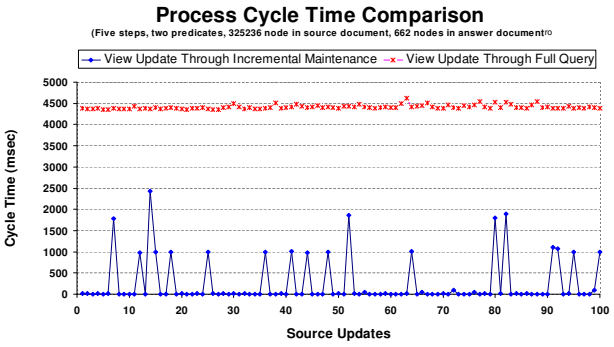
- *XP5*: /site/people/person[like(@id,"person2%")]/name/text()
- *XP6*: /site/people[person[like(@id,"person1%")]]/ person[like(@id,"person2%")]/name/text()

For each query, 100 source updates were randomly generated. The results of the time comparison for all the updates are shown in Figures 6(a) and 6(b). In short, full queries take 10 to 20 times longer to execute on average. The figures clearly establish the advantage of the proposed incremental view maintenance middleware.

Finally, consider Figure 7, which shows the caching impact analysis for query XP4, which has 13 steps, but no predicate. Since there are no predicates in XP4, no queries to the source need to be issued for predicate checking. Therefore, the time needed for incremental maintenance is rather constant, whereas the need for accessing sources for predicate tests had introduced a higher variability to the incremental maintenance time for queries XP5 and XP6 in Figures 6(a) and 6(b). Nevertheless, since predicate evaluation is only a part of the overall processing needed for reevaluation of queries XP5 and XP6, incremental maintenance was consistently cheaper even when sources are accessed for predicate checking.



(a) XP5



(b) XP6

Fig. 6. Incremental View Maintenance versus Full Re-Computation (Queries XP5, XP6)

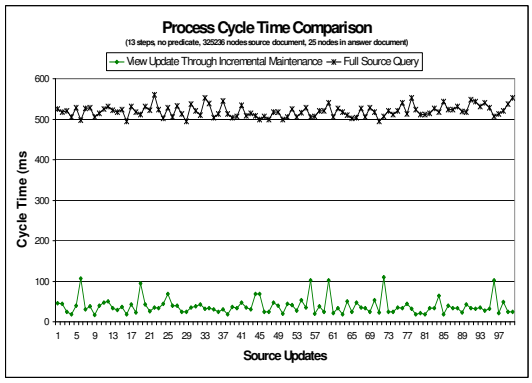


Fig. 7. Incremental View Maintenance versus Full Re-Computation (Query XP4)

6 Conclusion

In this paper, we have proposed WReX, a Web Service middleware architecture that enables cache management by bridging the gap between Web Service message caching and updates in the source data. Our solution consists of two components: (1) Web Service Content Description (WSCD) that fills the gap between Web Service messages and XML views of the source data; and (2) XML-specific view maintenance that fills the gap between XML views and updates in the source data. Cache-enabled Web Services are easily described and deployed on a common platform with proven RDBMS technology. Through experimental evaluation, we have demonstrated the performance benefits of our incremental view maintenance. Future work includes more effective maintenance of multiple XPath views and multiple updates, extension of our approach to other XML-to-RDBMS mapping schemes (such as schema-aware mappings), and more detailed studies on the entire middleware performance.

References

1. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD Conference*, pages 527–538, 2003.
2. K. S. Candan, D. Agrawal, W. Li, O. Po, and W. Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *The 28th Very Large Data Bases Conference*, 2002.
3. J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic web data. In *In Proceedings of IEEE INFOCOM'99*, 1999.
4. C. Y. Choi and Q. Luo. Template-based runtime invalidation for database-generated web contents. In *APWeb 2004*, 2004.
5. A. Datta, K. Dutta, H. M. Thomas, D. E. Vandermeer, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web:

An approach and implementation. *ACM Trans. Database Syst*, 29(2):403–443, 2004.

6. A. Deutsch, M. Fernandez, and D. Suciu. Storing Semi-structured Data with STORED. In *Proceedings of the 1999 ACM International Conference on Management of Data (SIGMOD'1999)*, 1999.
7. Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Syst*, 28(4):467–516, 2003.
8. D. Florescu, A. Grunhagen, and D. Kossmann. XL: An XML programming language for web service specification and composition. In *WWW2002, International World Wide Web Conference*, 2002.
9. D. florescu and D. Kossman. Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
10. Roy Goldman, Jason McHugh, and Jennifer Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Proceedings of the ACM International Workshop on the Web and Databases (WebDB'99)*, 1999.
11. J. Naughton, D. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy andJ. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, C. Zhang, B. Jacksonand A. Gupta, and R. Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2), 2001.
12. Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD Conference*, pages 903–908, 2004.
13. Arsany Sawires, Junichi Tatemura, Oliver Po, Divyakant Agrawal, and K. Selçuk Candan. Incremental Maintenance of Path-Expression Views. In *SIGMOD Conference*, 2005.
14. Albrecht Schmidt, Florian Waas, Martin L. Kersten, MichaelJ. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985, 2002.
15. Jayavel Shanmugasundaram, Rajashekhar Krishnamurthy, Igor Tatarinov, Eugene Shekita, Efstratios Viglas, Jerry Kinman, and Jefferey Naughton. A General Technique for Querying XML Documents using a Relational Database System. In *Proceedings of the 2001 ACM International Conference on Management of Data (SIGMOD'2001)*, 2001.
16. Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as xml documents. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB'2000), September 10-14, 2000, Cairo, Egypt*, pages 65–76, 2000.
17. Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 20002 ACM International Conference on Management of Data (SIGMOD'2002)*, pages 204–215, 2002.
18. D. B. Terry and V. Ramasubramanian. Caching xml web services for mobility. *ACM Queue*, 1(3):70–78, 2003.
19. K. Yagoub, D. Florescu, V. Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. In *The VLDB Journal*, pages 188–199, 2000.