Optimized XPath evaluation for Schema-compressed XML data

Stefan Böttcher, Rita Hartel, Stefan Heindorf
University of Paderborn (Germany)
Computer Science
Fürstenallee 11, D-33102 Paderborn
{stb@, rst@, heindorf@mail.}uni-paderborn.de

Abstract
XML has become the de facto standard for data exchange in enterprise information systems. But whenever XML data is stored or processed, e.g. in form of a DOM tree representation, the XML markup causes a huge blow-up of the memory consumption compared to the data, i.e., text and attribute values, contained in the XML document. In this paper, we present an optimized XPath query evaluation for XSDS, an XML compression approach based on removing information that is obsolete as this information can be derived from the existing XML Schema definition (XSD). Thereby, XSDS allows for storing and exchanging XML data in a space efficient and still queryable way. While previous papers have shown that XSDS generally reaches stronger compression ratios than other approaches like gzip, bzip2, and XMill and that XPath queries can be evaluated on XSDS compressed data, we show in this paper that when optimizing the query evaluation on XSDS compressed data by using the given schema information, we can speed up query evaluation by a factor of 13 reaching evaluation times that are more than 5 times faster than those of JAXP – the standard Java XPath evaluator. The speed up was reached by avoiding the decompression of large parts of the structure while evaluating the query.

Keywords: XML, XPath evaluation, XML compression

1 Introduction

1.1 Motivation
XML gains more and more popularity not only as a data exchange format but also as a storage, archive or data management format and XPath is the main standard to express path queries on XML data.

Whenever XPath query evaluation is a bottleneck of an application, a fast XPath query evaluator is desired. In addition, XML documents may become larger than the available main memory space. Therefore, it may be a significant advantage when the fast XPath query evaluator can process XPath queries on compressed XML documents that can still fit into main memory. Previous work ((Böttcher, Hartel & Messinger 2009), (Böttcher, Hartel & Messinger 2010), (Böttcher, Hartel & Messinger 2011)) has shown that XSDS shows stronger compression facilities than other generic data compressors and than other XML-specific compressors while at the same time allowing for query evaluation and direct search on the compressed data at a speed that is comparable to that of XPath query evaluation on the original uncompressed XML data. This can be achieved by using a generic XPath evaluation approach (e.g. (Böttcher & Steinmetz 2007)) that reduces query evaluation to navigation via the binary axes first-child, next-sibling and parent. By replacing this generic approach with a specific one that at the same time regards the additional knowledge that can be gained by the provided XML Schema Definition, we provide in this paper an XPath evaluator for XSDS compressed XML data that allows for evaluation times that are 13 times faster than the previously used generic approach and 5 times faster than query evaluation on uncompressed XML data using JAXP.

1.2 Contributions
In this paper, we present an optimized approach to XPath evaluation on XSDS compressed data that combines the following advantages:

- The approach allows for XPath evaluation directly on the compressed data, i.e., unnecessary decompression of the compressed data is avoided whenever possible.
- The approach uses the provided schema information to reduce the amount of compressed data to be tested during the evaluation process and allows for skipping large irrelevant parts of the document, whereas our previous approach had to parse at least the compressed document structure that comes in document order before the current context node.
- The approach suggests an additional index on the compressed data that does not significantly blow up the compressed data, but that allows for skipping complete sub-trees of the compressed XML document.

Combining all these properties, the presented approach allows for an efficient XPath evaluation on XSDS compressed data that is more than 13 times faster than the previously used generic approach for XPath evaluation on XSDS compressed XML data and up to 5 times faster than XPath evaluation on uncompressed XML data.

1.3 Query language
The subset of XPath expressions supported by our approach conforms to the set of core XPath as defined in (Gottlob, Koch & Pichler 2005). This set is defined by the following EBNF grammar:

\[
\begin{align*}
\text{cxp} & ::= '/\text{locationpath}' \\
\text{locationpath} & ::= \text{locationstep} ('/\text{locationstep})^*
\end{align*}
\]
Summary and Conclusions.

Our prototype with other XPath evaluators. Section 5 describes the index that allows for skipping sub-trees and the evaluation of XPath queries with filters. Furthermore, this section outlines some of the experiments that compare our prototype with other XPath evaluators. Section 5 gives an overview of related work and is followed by the

2 Compressing the data

2.1 The basic idea

The main compression principle of XML schema subtraction (XSDS) is to remove all information that is strictly defined by the XML schema definition from a given XML document, and to encode only those parts of the XML document in the compressed format that can vary according to the XML schema. In this paper, we only provide a short overview of XSDS compression, as details are described in (Böttcher, Hartel & Messinger 2009), (Böttcher, Hartel & Messinger 2010), and (Böttcher, Hartel & Messinger 2011).

2.2 This paper's example

To illustrate the ideas of our approach, we use an example of a schema, where each document consists of a <news>-element that (read from top to bottom) contains any number of <meta>-elements followed by any number of <article>-elements. Each <article>-element consists of any number of <meta>-elements, followed by a headline element, and any number of <content>-elements. Each <content>-element contains either a <txt>-element or an <img>-element. The elements <meta>, <headline>, <txt>, and <img> contain PCDATA only.

Fig. 1 shows a visualization of our example schema S called rule graph of S. The number in the lower part of each node represents the node ID of each node; the upper part represents the node’s label.

2.3 Compressing the Document

Compressing the structure. Within the structure of an XML document, i.e., within the element tags, there are only three different concepts that allow for variant parts within an XML document defined by a given schema: First, when the schema (XSD) requires the choice of one out of different given alternatives. Second, when the XSD element ‘all’ requires the occurrence of all elements declared by children of the ‘all’ element, but they can occur in any order. Third, when the XSD requires a repetition of elements, which allows for a varying number of elements (including all its descendant elements).

The compression of these variant parts within an XML document works as follows. Each compression step assumes that we consider one current position in the XML document at a time for which the XSD allows variant parts. For each current position in the XML document for which the XSD allows a choice, we only store the alternative chosen at this current position. For each XSD element ‘all’, we only encode the order of the elements required by the children of the ‘all’ element in the XSD. Finally, for each repetition of elements starting at a given position within an XML document, we only store the number of occurrences of this element found at the current position of the XML document.

Compressing the Textual Data. Beneath the structure, an XML document contains textual data. We store the text data in document order in a text container and apply gzip on top of the container at the end of the document.

3 Optimized query evaluation

3.1 The basic idea

The goal of our optimized query evaluator for schema-compressed XML data is to avoid considering and thereby decompressing parts of the compressed document that cannot contribute to the query evaluation, i.e., we want to skip those parts of the compressed document D that do not need to be read to answer the given query.

As for XSDS the existence of an XML schema is required, we use the information given by the schema’s rule graph to optimize the evaluation of a given XPath query Q as follows. For each rule graph node, we determine whether or not it is relevant to Q, in such a way that those parts of D that correspond to non-relevant rule graph nodes can be skipped.

Let us consider for example the query //meta. Without additional knowledge, the complete document would have to be read, as an element node with label ‘meta’ could occur at any depth within the document. But if we consider the schema shown in Fig. 1, we know for example that there does not exist an element node with label ‘meta’ that is a descendant of a node with label ‘content’. Therefore, we can skip the sub-tree below the content node and continue the query evaluation with the next-sibling of the content-node.

In order to benefit from the XML schema definition given in the schema’s rule graph within the query evaluation, intuitively, we determine for each location...
3.2 An optimization step extending the rule graph

Whether or not a next rule graph node is relevant to a location step of an XPath query depends not only on the current rule graph node, but on the path through the rule graph to the current rule graph node taken by previous location steps. That is, in general, relevance of a next rule graph node cannot be decided locally depending on a current rule graph node, and therefore, whether or not a next XML node has to be read cannot be decided locally on the current XML node and current rule graph position.

In order to further reduce the relevant rule graph nodes and thereby reduce the parts of the compressed XML document to be read, we do the following preprocessing step before building the navigation guide. We extend the rule graph, such that whether or not a rule graph node is relevant to a location step of an XPath query can be determined locally on the extended rule graph (ERG) node, i.e., independent of the path via which the ERG node was reached when evaluating the XPath query.

The extended rule graph ERG then fulfills the unique successor set property that each element node en of ERG uniquely defines the set ENAX of ERG nodes that can be reached via any of the axes \( \text{ax} \in \{\text{self}, \text{child}, \text{descendant-or-self, descendant}, \text{following-sibling}\} \) of \( \text{en} \), i.e., ENAX = \( \{\text{enax} | \text{enax} \in \text{en/ax}\} \) is not larger than the set of nodes PENAX that can be reached by a certain path \( P \) from the root of ERG via \( \text{en} \) using the axis \( \text{ax} \) with PENAX = \( \{\text{penax} | \text{penax} \in \{P/\text{en}/\text{ax}\}\} \).

For example, this unique successor property does not hold for the element-node \( \langle \text{meta} \rangle \) of the rule graph given in Fig. 1, as this element has different following-sibling nodes depending on whether it has the parent node \( \langle \text{article} \rangle \) or the parent node \( \langle \text{news} \rangle \), i.e., //article/meta/next-sibling::* is a \( \langle \text{headline} \rangle \) element or a \( \langle \text{meta} \rangle \) element, however //news/meta/next-sibling::* is an \( \langle \text{article} \rangle \) element or a \( \langle \text{meta} \rangle \) element. Note that the axes self, child, descendant-or-self and descendant following-sibling already fulfill the unique successor set property, and only the axis following-sibling does not always fulfill the unique successor property, i.e., the rule graph is extended to have the unique successor property fulfilled for the following-sibling axis.

Whenever a rule graph node \( n \) does not fulfill the unique successor set property, we split the node \( n \) and its incoming paths in RG into a minimum number of nodes \( n_1, \ldots, n_p \) and a minimum number of incoming paths to \( n_1, \ldots, n_p \) in ERG as follows, such that each of the nodes \( n_1, \ldots, n_p \) fulfills the unique successor set property.

Splitting a node \( n \) and its incoming paths in RG means that in RG is replaced with \( n_1, \ldots, n_p \) in ERG, and the incoming paths to \( n \) are replaced by incoming paths to \( n_1, \ldots, n_p \) in ERG, such that each path in ERG to \( n_1 \) or \( n_p \) has a corresponding path in RG and vice versa, where a corresponding path has the same number of nodes and edges and the same sequence of node labels. Splitting a node \( n \) in RG into nodes \( n_1, \ldots, n_p \) in ERG includes that for each node \( n_k \) in ERG, each edge \( (n, ns) \) from \( n \) to a successor node \( ns \) in RG is copied to an edge \( (n_k, ns) \) in ERG. The remaining rule graph (e.g. c.f. Fig. 2) then fulfills the unique successor set property.

![Fig. 2. Extended rule graph (ERG) of the rule graph of Fig. 1 after the preprocessing](image)

Note that the ERG is not necessarily a tree, but is a graph that still may contain cycles. Fig. 3 shows on the left-hand side an example of a (simplified) graph that contains cycles and does not fulfill the unique successor set property for the nodes with label ‘e’ and ‘f’, and it shows on the left-hand side an extended graph that fulfills the unique successor set property although it still contains cycles.

![Fig. 3. Example of a recursive graph and the corresponding extended graph fulfilling the unique successor set property](image)

3.3 Constructing the Navigation Guide

The navigation guide shall contain all the paths through ERG that correspond to an answer to the query for at least one legal XML document. In order to construct the navigation guide, we “simulate” the query evaluation on ERG location step by location step, but without any access to or knowledge about the real compressed data.

Within a pre-processing step using the looking forward approach, each XPath query is transformed into an equivalent XPath query \( Q \) that does not contain any backward axes. Although \( Q \) does not contain any backward axes, for simulating on ERG the execution of a following-sibling axis contained in \( Q \), we have to navigate backwards and forward again on ERG. In order to illustrate and to execute the required navigation steps, we construct a data structure called navigation guide from the ERG. The navigation guide NG for a given XPath query \( Q \) is constructed from \( Q \) and the ERG in two consecutive blocks of steps.

Computing the nodes being reachable by the query. In the first part of our approach, we copy paths of nodes...
from ERG to corresponding paths of nodes in NG for each location step LSi of the query Q. We say that a path to a node n in NG is corresponding to a path to a node e in ERG and we call the node n corresponding to e, if both paths have the same sequence of node names, ignoring the direction of edges in the graphs of ERG and NG. Paths through the ERG are copied to corresponding paths in NG for only those ERG nodes which define the XML nodes that are accessed when LSi is evaluated.

For example, if we consider the ERG given in Fig. 2 and the query //meta, we find the paths 1-2-3-4 and 1-2-6-7-8-9-4' through ERG for the location step //meta. These paths are copied to the NG.

Furthermore, although NG may contain cycles for recursive schemas, NG grows like a tree, i.e., common prefixes of paths in NG are stored only once, and paths copied to NG are appended to NG’s leaf nodes.

In the example, the common prefix is 1-2, and the leaf nodes of NG are 4 and 4'.

More precisely, at the beginning, NG consists only of a single node that is copied from the root node of ERG, the node 1 in the example.

Then, repeatedly for each location step LSi, NG is modified depending on the axis of the current location step LSi=axis::nnt as follows.

If the axis is the self-axis, the leaf nodes n in NG not corresponding to an ERG node e that conforms to the node name test nnt and the incoming edges to n are deleted from NG.

Except for self-axis location steps, NG is extended by copying ERG paths to corresponding NG paths and attaching the copies of paths starting in an ERG node e to the corresponding leaf node n of NG.

If the axis is the child-axis, for each leaf node n in NG the corresponding node e in ERG is determined, and each path in ERG from e to a next element node e2 in ERG not containing any other element node except of e and e2 is determined. For each element node e2 that fulfills the given node name test nnt, the path from e to e2 is copied to NG, i.e. appended to n.

If the axis is the descendant-axis, for each leaf node n in NG the corresponding node e in ERG is determined, and each path in ERG from e to any further element node e2 in ERG is determined. For each element node e2 that fulfills the given node name test nnt, the path from e to e2 is copied to NG, i.e. appended to n.

If the axis is the following-sibling-axis, for each leaf node n in NG the corresponding node e in ERG is determined. Starting from e, those element nodes e2 in ERG are determined that define a following-sibling of the XML elements defined by e. To find a connection from e to these element nodes e2 in ERG, we have to consider the semantics of the “( , , )”, “|”, and “*” nodes. Then, connections from e to e2 are copied to paths in NG for each element node e2 that fulfills the given node name test nnt, even if the connection uses edges of ERG in reverse direction.

For example, to find a connection in ERG to a next-sibling <headline> element e2 when starting in the <meta> element e represented by node 4' in Figure 2, we have to pass the nodes 4'-9-8-10, i.e. we have to go from 4' via 9 to 8 in opposite direction of the edges found in ERG. Nevertheless, the path copied to NG contains copies of the nodes 4', 9, 8, and 10 in this order.

To extend the example, if we consider the rule graph given in Fig. 2 and the query //meta/following-sibling::headline, we find the paths 1-2-3-4 and 1-2-6-7-8-9-4' for the location steps //meta. Starting from the node 4, we do not find a result node for the next location step following-sibling::headline, but starting from the node 4', we find the path 4'-9-8-10. Note that the corresponding edges in the NG go from 4' to 9 and from 9 to 8 as navigation in ERG to the following-sibling follows this path, although the corresponding edges in the ERG shown in Fig. 2 take the opposite direction.

Deleting nodes that cannot contribute to the query result: In a second block of steps, we do a depth-first search backwards from the NG nodes representing XPath query result nodes to the root node of NG and mark all visited nodes. Thereafter, we eliminate from NG all those nodes that are not marked as they lead to a result of one of the intermediate location steps that cannot contribute to a result of the query. The paths remaining after the second step form the final graph structure of the navigation guide.

Continuing the example, the input of our second block of steps consists of the sub-graph containing the paths 1-2-3-4 and 1-2-6-7-8-9-4'-9-8-10. The result of the depth first search backwards that uses the edges of the NG in opposite direction starting at node 10 (the single result node) does not visit the nodes 3 and 4, i.e. the nodes 3 and 4 are deleted. Thus, the final NG shown in Fig. 4 only contains the path 1-2-6-7-8-9-4'-9-8-10. Here, the sub-path 1-2-6-7-8-9-4' conforms to location step //meta and the sub-path 4'-9-8-10 conforms to location step following-sibling::headline.

![Fig. 4. Graph structure of the navigation guide for query //meta/following-sibling::headline](image)

Note that more than one copy of the same ERG node e might occur in the navigation guide as e is visited by different location steps, e.g. two copies of each of the nodes 8 and 9 occur in NG.

3.4 Evaluation of path queries without filters
The evaluation of a query based on the extended rule graph (ERG) and on the navigation guide (NG) is similar to the decompression of the document combined with a skipping of irrelevant sub-trees. This means, while we sequentially process compressed XML data, we walk synchronously in pre-order through the ERG and through
the NG. Whenever we pass a ‘*’-node in ERG, we determine how often to walk through its ERG sub-tree, and whenever we pass a ‘?’-node in ERG, we determine which path in ERG to follow. But in contrast to the decompression of XSDS compressed XML data based on ERG alone, we follow only those ERG paths that have a corresponding path in NG, and thereby skip large, irrelevant parts of the compressed document.

In addition to the graph structure of the navigation guide, we maintain a stack S that contains sets of navigation guide nodes and that represents the walk through the navigation guide and is handled synchronously to the recursion stack of the pre-order walk through the ERG. We start the query evaluation at the beginning of the compressed data and in the root fn of the ERG and in the root fn of NG (where fn corresponds to fe), and the stack S contains only one set containing the first node fn of the navigation guide NG.

Whenever the recursive pre-order walk through the ERG requires following an edge from a node e1 to a node e2, where e1 corresponds to a node n1 in the set on top of the stack S, we check, whether there exists an edge in NG from n1 to an NG node n2, such that e2 corresponds to n2. If at least one such node n2 ∈ NG exists, we proceed to n2 in the ERG and push to S the set NG2 that contains all nodes n2 ∈ NG to which cn2 corresponds and that are children of any node n1 ∈ NG that was on top of stack. If there is no such edge in the navigation guide, we skip the sub-tree beyond this edge in the ERG.

For example, consider that we are currently in node 2 of ERG shown in Fig. 2, and the stack S contains the two sets [{//meta:1}, {/meta:2}], i.e., the set at top of S contains only the node with ID 2 shown in Fig. 4 as one of the NG nodes generated for the location step /meta. The first outgoing edge of node 2 in ERG is the edge E1=(2,3). But as there is no corresponding edge (/meta:2,x) in NG, we can skip the sub-tree rooted by node 3 in ERG. Instead, we check the next outgoing edge E2=(2,6) of node 2 in ERG. As we find the corresponding edge (/meta:2,6) in NG, we follow the edge E2 in ERG and push a set containing the node //meta:6 to the stack S, i.e., we get S=[ {/meta:1}, {/meta:2}, {/meta:6}].

Whenever the recursive walk through ERG requires a backtracking step (i.e. a pop-operation on the recursion stack) such that we go back from a node en2 to a node en1 in the ERG, we perform a pop-operation on S as well. In addition, we add the set NG1 to the top-of-stack that contains all nodes n1 that correspond to such a node en1 and that are children of any node n2 that was on top of stack.

If we perform for example the backtracking from node 4’ to node 9 in the ERG, we pop the entry /meta:4’ (for the node of location step /meta with ID 4’) from the stack, such that the entry //meta:9 becomes the new top-of-stack. In addition, we add the entry /following-sibling::headline:9 to the current top-of-stack, as there is an edge from //meta:4’ to /following-sibling::headline:9 in the navigation graph, such that the new top-of-stack consists of the entries //meta:9 and /following-sibling::headline:9.

Whenever the result node of the navigation guide (i.e. the node /following-sibling::headline:10 in our example) is stored on the stack, the current position within the compressed data is a result to the query.

3.5 Skipping sub-trees
As the compressed data generated by XSDS does not allow for skipping sub-trees of a compressed XML document without additional information, we have extended the XSDS compression in such a way that it inserts a small skip-index into the compressed data. This index stores the address within the compressed data of the end of each repeated XML sub-tree structure that conforms to a sub-tree of the ERG that has a repetition node with label ‘*’ as root. Such an address consists of a position in the compressed XML structure and a position in the compressed text values of the XML document. Whenever the navigation guide allows for skipping a sub-tree with root node rn in the ERG, we walk through the ERG and the compressed representation recursively until we reach rn a second time (within the backtracking). Whenever we find a repetition node R in ERG within this skipping process, we read the address of the end of the compressed data belong to R and directly jump to that end position in the compressed structure and in the compressed text data, thereby skipping large parts of the compressed document.

3.6 Evaluation of queries with filters
Concerning the evaluation of predicate filters, we follow the idea of (Böttcher & Steinmetz 2007): Whenever a location step contains one or more predicate filters, we generate and activate a navigation guide FNG for the filter path. At the same time, we add a so-called reservation to the current entries of the set CS on top of the stack. This reservation is connected to FNG and ensures, that the entries of CS are considered valid, if and only if FNG reaches a final state, i.e., if and only if the filter can be evaluated to true for these nodes. Each filter navigation guide has the same principal design and functionality as the navigation guide of the main path of Q as described in the previous sections. Only if the execution of the filter yields true for a given stack entry, i.e., if the result node of the filter navigation guide is reached, the reservation is deleted and the entry is considered invalid. If the execution of the filter navigation guide yields false for this entry, the entry itself is deleted and considered invalid.

The location-steps of the filter navigation guide can be connected to other filter navigation guides, such that nested XPath filter expressions can be evaluated by this concept as well.

4 Evaluation of our Prototype Implementation
4.1 Evaluation environment
We have compared the performance of our optimization to the performance of the previously used generic XPath evaluator (Böttcher & Steinmetz 2007), called XMLFramework, to the Java standard evaluator JAXP and to the state of the art approaches SAXON (SAXON.
The XSLT and XQuery Processor n.d.) and eXist (eXist: An open source native XML database n.d.). For this purpose, we used the following queries from the XPathMark performance benchmark (Franceschet 2005):

- Q1: /site/closed_auctions/closed_auction/annotation/description/text/keyword
- Q2: //closed_auction//keyword
- Q3: /site/closed_auctions/closed_auction/keyword
- Q4: /site/closed_auctions/closed_auction[annotation/description/text/keyword]/date
- Q5: /site/closed_auctions/closed_auction[descendant::keyword]/date
- Q6: /site/open_auctions/open_auction/bidder[following-sibling::bidder]

The following documents generated with XMark (Schmidt et al. 2002) served as our test documents (each document name is of the form d.<factor>.xml, where ‘factor’ is the /f parameter passed to the XMark generator):

- d0.001.xml
- d0.010.xml
- d0.100.xml
- d1.000.xml

Our tests were executed on an Intel Core 2 Duo 1.8 GHz with 2 GB of RAM. We used Microsoft Windows XP SP 3 with Java SE 6 Update 23. The java heapsize was set to 1024 MB. We used version 9.3 of Saxon and 1.4.0 of eXist. We ran at least 5 consecutive executions of each query.

XMLFramework and XXE were executed directly on the XSDS compressed documents - the XMLFramework on the version without the skip index and XXE on the version with skip index. JAXP and SAXON were executed on the original, uncompressed documents, and eXist uses its own dedicated database with indices.

### Table 1. XMark documents and representation sizes used in our evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Original Size</th>
<th>Size XSDS (w/o index)</th>
<th>Size XSDS + index</th>
<th>eXist(^1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d0.001.xml</td>
<td>0.11 MB</td>
<td>0.04 MB</td>
<td>0.04 MB</td>
<td>0.16 MB</td>
</tr>
<tr>
<td>d0.010.xml</td>
<td>1.13 MB</td>
<td>0.36 MB</td>
<td>0.36 MB</td>
<td>1.64 MB</td>
</tr>
<tr>
<td>d0.100.xml</td>
<td>11.32 MB</td>
<td>3.48 MB</td>
<td>3.59 MB</td>
<td>18.21 MB</td>
</tr>
<tr>
<td>d1.000.xml</td>
<td>113.06 MB</td>
<td>34.79 MB</td>
<td>35.94 MB</td>
<td>183.55 MB</td>
</tr>
</tbody>
</table>

\(^1\) Measured as the increase in size of the data folder with journaling turned off.

## 4.2 Evaluation results

The evaluation results are shown in Fig. 5 and 6.

XXE outperforms the XMLFramework, JAXP and SAXON, whereas the advantage of using XXE gets larger with increasing document size (although all approaches scale linearly). Only eXist is faster by using larger indices and thus, causing higher storage requirements. Furthermore, XXE is clearly faster than completely decompressing the document, e.g. the decompression of d1.000.xml takes 37 seconds on our test machine.

### 5 Related Works

There exist several different approaches to the evaluation of XPath queries on XML data streams. Nearly all of them are based on automata (X-scan(Ives, Halevy & Weld 2002), YFilter (Diao, Rizvi & Franklin 2004), (Green et al. 2004), (Gupta & Suciu 2003), AFilter (Candan et al. 2006)) or parse trees ((Bar-Yossef, Fontoura & Josifovski 2007),(Barton et al. 2003), (Chen, Davidson & Zheng 2006)). All of them support the axes child and descendant-or-self and most of them support predicate filters and wildcards, but besides (Onizuka 2010) none of them support the sibling-axes as our solution does. All of these approaches require plaintext XML as input, typically in form of a SAX event stream. Applying these approaches to compressed XML would lead to a complete decompression of the compressed XML document into SAX events.

(Böttcher & Steinmetz 2007) and (Benter, Böttcher & Hartel 2011) are generic approaches that can be applied to XML representations that allow – similar as in this approach – navigation along the basic axes first-child,
next-sibling and self plus parent in case of (Böttcher & Steinmetz 2007) and the inverse axes of first-child and next-sibling in case of (Benter, Böttcher & Hartel 2011). Therefore they can be applied to plain text XML as well as to compressed XML without forcing a complete decompression of the compressed XML data. (Arroyuelo et al. 2010) and (Maneth & Nguyen 2010) present a compressed representation for XML together with an XPath evaluator that is based on tree automata and that allows to skip irrelevant parts of the compressed XML document during the evaluation process. They allow selecting a single start point and follow the path to the root bottom-up and the path to the “leafs” of the query top-down.

Bisimulation (Buneman et al. 2005) and XQueC (Arion et al. 2007) are further approaches that allow the evaluation of XQuery instructions on top of compressed XML data. XQueC (Arion et al. 2007) proposes an XML representation that is optimized for an efficient transformation via XQuery. The structure compression as well as the data compression is chosen in such a way that path queries can be evaluated efficiently on it, but in return, the compression ratio reached by XQueC (Arion et al. 2007) is not as strong as by other compressors. The approaches being used in XQueC (Arion et al. 2007) appear to be non-applicable to other compression techniques. A document that is compressed via Bisimulation can be transformed with the help of XQuery into a compressed target document. In contrast to our approach, Bisimulation (Buneman et al. 2005) supports XPath expressions consisting of child axis steps only, whereas our approach supports all forward axes except the following axis.

6 Summary and Conclusions

Whenever the combination of XPath query evaluation with high storage costs or with low data transfer rates is a bottleneck of an application, using XSDS compressed data instead of the original XML data might be a solution to overcome this problem. We propose an optimized query optimization for XSDS compressed data that allows for XPath evaluation directly on the compressed data, i.e., unnecessary decompression of the compressed data is avoided whenever possible. Furthermore, the approach uses the provided schema information to reduce the amount of compressed data to be tested during the evaluation process and allows for skipping large irrelevant parts of the document. In order to allow the skipping of these irrelevant parts, the approach suggests an additional index on the compressed data that does not significantly blow up the compressed data.

Our experiments have shown that our approach outperforms previous approaches that allow the query evaluation on XSDS compressed data, and furthermore, it even outperforms the Java standard evaluator JAXP and the state of the art approach SAXON. In our experiments it was outperformed only by eXist. But in contrast to eXist, which needs more than 150% storage space of the original data, our approach only requires 32%, i.e., XXE decreases the storage and transfer costs, whereas eXist even would increase them.

7 References


Diao, Y, Rizvi, S & Franklin, MJ 2004, 'Towards an Internet-Scale XML Dissemination Service', *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, Morgan Kaufmann, Toronto, Canada.


Onizuka, M 2010, 'Processing XPath queries with forward and downward axes over XML streams', *EDBT 2010, 13th International Conference on Extending Database Technology*, Lausanne, Switzerland.
