Abstract
Within XML data streams, markup as defined e.g. in a DTD is not only being used for structuring large amounts of data, but also for efficiently searching, accessing, and processing the required parts of the data streams. However when huge amounts of XML data are involved, data reduction or compression techniques that still allow finding the required parts of the data fast may become crucial to handle data processing.
We present a data reduction and compression technique for XML data streams that not only significantly reduces the amount of data, but also allows for efficient data processing without requiring a full data decompression. Our data reduction technique combines sub-tree sharing with removing structure that is known by a DTD. We have done extensive performance evaluations to compare our compression technique with other approaches to XML compression, and we show that we not only outperform the other techniques, but also outperform string compression techniques like gzip that do not support query processing on compressed data.

1. Introduction

XML is a widely used standard for data exchange in the web, and data sources like publish/subscribe systems produce – according to [7] – several GB of XML data per minute that are structured according to a given DTD. As most web standards are based on XML, we expect that the amount of XML data will even be significantly increased in the near future, and querying, exchanging and processing huge amounts of data will be one of the major challenges in massive web data processing. Data reduction techniques like XML compression have the goal to reduce the data volume by magnitudes providing significantly faster exchange of XML data without limiting the ability to search, query, or process the compressed data.

1.1. Classification of related work

Previous work on XML data reduction can be classified mainly into lossy and lossless data reduction techniques. While lossy data reduction does not allow for a precise reconstruction of the data and may use approximate answers instead of exact answers to queries, we follow the lossless approach to XML data reduction which allows a full reconstruction of arbitrary large parts of the compressed data and which supports exact answers to queries. Lossless XML data reduction approaches use two major approaches: XML structure encodings and XML compression techniques. XML structure encodings use symbols for all element names, and some of them, like XGrind [15], [9] and BSBC [2], use a bit-stream to represent the structure of an XML tree, whereas other structure encodings, like XMLPPM [5] depend on the probability of the occurrence of certain elements. Within the XML compression approaches, we can distinguish ‘non-queryable compression techniques’ as XMill [11] that produce non-queryable compressed data from ‘queryable XML compression techniques’ that produce queryable compressed XML data. We favor queryable XML compression techniques, as they avoid full decompression for search, query answering or processing the compressed XML data. Within queryable XML compression, the following main directions can be distinguished:

Schema-based compression techniques rely on an XML schema or a DTD and remove information from XML data that can be deduced from the schema or from the DTD. They do not encode the elements in the XML document, the occurrence of which is granted by the DTD or the schema. Examples for schema-based compression techniques are XCQ [12] and Xenia [16] that use automata-based approaches to reduce the data volume, and DTD subtraction [3] that counts repetitions of sibling elements.

Structure sharing-based compression techniques, e.g. [4], XQZip [6] and LZCS [1], identify common sub-structures of XML data and use pointers to
common sub-trees instead of repeating XML data. They work on directed acyclic graphs (DAGs) rather than on XML trees such that only the first occurrence of each common sub-tree has to be encoded.

While schema-based compression yields multiple identical chunks of compressed data for sub-trees that occur multiple times in the XML data as it does not take advantage of common sub-trees, common sub-structure based approaches to XML compression avoid duplicate storage of common sub-trees, but they often perform even poorer as they do not use the information provided by a DTD or XML schema. Up to now, there has not been a known way how to combine the advantages of both approaches and to avoid the disadvantages.

1.2. Contributions

Our paper has the following contributions:

• We extend and significantly improve the encoding technique for the DTD subtraction leading to a very succinct signature of the tree structure of the XML data, called KST signature.

• We propose to separate the pointer implementation for sharing of compressed XML data into two parts: the part encoding the origin, and the part encoding the destination. For both parts, we present different implementation techniques.

• We have developed a hybrid approach to XML compression that applies DAG-like structure sharing to DTD subtraction. This includes a technique for identifying XML sub-trees that are suitable for structure sharing of their signature.

• We have conducted extensive performance evaluations that show that our approach has a comparable query performance and significantly outperforms all the other approaches to XML compression.

2. Reducing the structure of XML files

We follow the commonly used approach (c.f. e.g. [2,3,9,12,16]) in XML compression of separating constant compression from structure compression for the following reasons.

• We get better overall compression results and significantly better compression ratios for structure.

• The structure is often sufficient for navigation to certain content.

2.1. Overview of the KST signature generation

The compressed structure index produced by our compression approach is called KST signature in the remainder of this paper. KST signature generation applied to an XML document removes all the structural information from the XML document which is redundant, as it can be derived from the DTD of the XML data, and it stores the remaining structural information of the XML document in the so called KST signature.

The DTD is a regular tree grammar describing XML trees. As such, it can not only be used for parsing the XML tree, but also for generating a KST signature as a ‘side-effect’ of parsing the XML tree. For this purpose, the regular tree grammar of the DTD is augmented to an attribute grammar that generates KST output during the parsing process.

A special feature of this KST signature generation is that the issued KST signature, can also be parsed using the same DTD rules. Furthermore, when the KST signature is parsed using the attribute grammar extending the DTD rules, the original XML document can be generated – simply by inverting the substitution operations of the attribute grammar that generated the KST signature.

More details of concrete algorithms on KST signature generation are given in [3].

2.2. Attribute grammar-based KST signature compression by DTD subtraction

Which structural information of an XML document can be derived from the DTD, depends on the regular expressions used in the right-hand side of the DTD’s element declarations and on their specific DTD operators ‘|’, ‘,’ ‘?’ ‘*’, ‘+’, PCDATA, EMPTY, and E where E is an element name. More precisely, for each of these operators, DTD subtraction defines how a regular expression composed of these operators and smaller regular expressions can be compressed to the KST signature. Let ‘r’ and ‘s’ be regular expressions occurring in the right hand side of an element declaration (c.f. Table 1 (a)) and let R and S be sub-trees of an XML document corresponding to ‘r’ and ‘s’, then the KST signature can be computed recursively when using the DTD to parse the XML document roughly as follows (for details c.f. [3]):

• If the regular expression ‘r’ describing the sub-tree R to be parsed next consists of an element name E only, there is no choice for this sub-tree R of the XML document, as the root of R must be the element ‘E’. As this information can be derived from the DTD, the element ‘E’ can be removed from the XML document when the KST signature is generated and nothing is generated in the KST signature to represent the element E.

• Similarly, if the expression ‘r’ describing the sub-tree R to be parsed next is PCDATA or is EMPTY,
there is no choice for the sub-tree R of the XML document, as R must contain a text value or must be empty respectively. Therefore, nothing is generated in the KST signature to represent this information.

- Each regular expression ‘r, s’ occurring in a DTD element declaration, determines explicitly that a sub-tree R corresponding to ‘r’ has to be followed by a sub-tree S corresponding to ‘s’ in every valid XML document. Again, there is no alternative to S matching ‘s’, when R matching ‘r’ has been parsed, and no further information has to be encoded in the KST signature. For example, the DTD of Table 1(a) already defines, that the element ‘i’ has a first child ‘g’, followed by a second child ‘h’. Therefore, nothing has to be encoded for representing the elements ‘g’ and ‘h’ (in lines (3) and (4) of the XML document shown in Table 1(b)) in the KST signature shown in Table 1(c).

- Each regular expression ‘r | s’ occurring in a DTD element declaration, determines that either a sub-tree R corresponding to ‘r’ or a sub-tree S corresponding to ‘s’ follows next within the XML document. Which alternative follows in the XML document, can not be derived from the DTD. Therefore, which alternative follows, has to be encoded in the KST signature.

- The expression ‘r?’ defines that either one or zero XML sub-trees R that correspond to ‘r’ follow next within the XML document. The information whether or not a sub-tree R corresponding to ‘r’ follows in the XML document is encoded in the KST signature by a ‘1’ or a ‘0’ respectively.

- The regular expressions ‘r*’ and ‘r+’ define that any number of XML sub-trees R that correspond to expression ‘r’ (or at least one such sub-tree in the case of ‘+’) follow next in the XML document. As the KST signature compression of ‘r’ is performed as described before, the only remaining information to be written to the KST signature is the number x (or x-1 in the case of ‘+’) of sub-trees R corresponding to ‘r’ that follow in the XML document.

Example: Table 1 shows (a) a DTD, (b) an XML document and (c) the resulting KST signature. As the DTD defines that the element ‘e’ contains any number of ‘f’-children, the concrete number 2 of ‘f’-children can not be deduced from the DTD and is thus encoded in the KST signature. However, the DTD defines that every ‘f’ element has exactly one ‘g’ child and exactly one ‘h’ child, thus the occurrence of these elements does not need to be encoded in the KST signature. Finally, as according to the DTD, each ‘i’-element either has or does not have an ‘i’-element as its third child, the existence or non-existence of each possible ‘i’-child of an ‘f’ element is encoded by ‘1’ or by ‘0’ in the KST signature. This is why the KST signature in Table 1(c) of the XML file in Table 1(b) and the DTD in Table 1(a) is

\[
\begin{align*}
\text{DTD} & : \text{Table 1 (a) DTD, (b) XML document and (c) KST signature} \\
\text{(a)} & : \text{<!ELEMENT (EMPTY)>} \\
\text{(b)} & : \text{<!ELEMENT e (f*)> (1) <e> (2) <f> 2} \\
\text{(c)} & : \text{<!ELEMENT f (g,h,i?)> (3) <g/> (4) <h/> 1} \\
\text{<!ELEMENT g (EMPTY)> (5) <i/> 1} \\
\text{<!ELEMENT h (EMPTY)> (6) </f> (7) <f> 1} \\
\text{<!ELEMENT i (EMPTY)> (8) <g/> (9) <h/>} \\
\text{<!ELEMENT e (f*)> (10) </i/>} \\
\text{<!ELEMENT f (g,h,i?)> (11) </f> (12) </e> 1} \\
\end{align*}
\]

\[2 1 1\]

Attributes can be encoded like child elements, and attribute declarations have to be treated analogously to element declarations. However, to simplify the presentation, we focus on elements.

2.3. Improvements in KST signature encoding

The compression result KST generated by DTD subtraction [3] uses one byte for each encoded token within the KST, except for very large numbers, for which an overflow encoding is provided that allows for using n Bytes, where n is at least 1 and depends on the value of the number to be encoded.

We have improved the KST signature encoding for the regular expressions mentioned above as follows:

- For a regular expressions ‘r | … | s’ occurring in the DTD, the KST signature needs to encode, to which of the provided alternative regular expressions ‘r’ or ‘s’ the sub-tree (R or S) found at actual position of the XML document corresponds. As the number n of different alternatives can be derived from the DTD, we encode this information with \(\lceil \log_2 n \rceil\) bits within the KST signature.

- For the regular expression ‘r?’ occurring in the DTD, the KST signature needs to encode whether or not a sub-tree R matching ‘r’ occurs at the current context node of the XML document. We use one bit for encoding this in the KST signature.

- For the regular expressions ‘r*’ and ‘r+’ occurring in the DTD, the KST signature needs to encode the number of occurrences of sub-trees Ri in the XML document that correspond to ‘r’. To improve this encoding, we have performed a series of measurements on many different XML documents with the result that the frequency distribution of these numbers of occurrences is quite similar over all
evaluated XML documents. Therefore, we have calculated a DTD-independent static Huffman encoding for these numbers of occurrences which includes an ‘overflow encoding’ for all the ‘high numbers’ of occurrences. Due to these optimizations, we need only 0.7*n bits on average to encode the n occurrences of sub-trees R matching ‘r’, in comparison to the previously best other compression approaches, [12] and [16], that need n bits.

Applying these optimizations to the encoding of the KST already resulted in a significantly better compression ratio within the KST signature. In order to improve the compression ratio even more, we propose to combine the DTD-based compression with a DAG-based compression, e.g. [4], as described in the following sections.

3. Combined DAG and KST compression

A compression approach that is solely based on sub-structure sharing is described in [4]. However, combining structure sharing with KST signature compression is a challenging task because not every sub-structure of the KST signature is suitable for structure sharing.

In this section, we first give an example of a KST signature sub-structure that is not suitable for structure sharing, and then describe which sub-structures of the KST signature are suitable for structure sharing.

3.1. KST signature sub-structures unsuitable for structure sharing

DAG compression of XML trees either works on unranked XML trees or on binary XML trees. We follow the binary DAG compression approach.

In the example of the XML document of Table 1(b), the second binary sub-tree ‘<g/><h/><i/>’ in lines (8)-(10) is equal to the first one in lines (3)-(5). Therefore, in a binary DAG of the XML document, the sub-tree in lines (8)-(10) would be replaced with a pointer to line (3). However, the ‘f’ elements are not shared, as they do not have equal next siblings.

When trying to implement the corresponding pointer to line (3) within the KST signature, we would have to identify the position of the pointer origin in the KST signature, which is after the first ‘1’. But as this position could refer to the ‘f’-element of line (7) as well as to the ‘g’-element of line (8), the position of the pointer origin in the KST signature is ambiguous. In other words, a unique decompression of a KST signature using this pointer would be impossible.

3.2. How and when are KST signatures re-used

In order to avoid the ambiguities described above, we only allow for KST signature structure sharing using pointer origins which are unambiguous as the pointer’s origins directly correspond to a KST signature position. Such a unique mapping of XML nodes to KST signature positions exists for so called explicit XML nodes.

Definition: An explicit XML node en is a node of the XML document, for which a KST token has been generated, i.e. no other node has been parsed between the parsing of en and the encoding of a KST token.

For example, the nodes in lines (2), (5), and (10) of the XML document of Table 1 (b) are explicit XML nodes, but the node in line (7) is not an explicit XML node, as before encoding of the next KST token ‘1’, the nodes in lines (8)-(10) are parsed.

When using structure sharing on explicit XML nodes, we can uniquely identify the KST signature token corresponding to an explicit node. In other words, we only use DAG pointers that start after an explicit XML node because thereby pointer origins correspond to a KST signature position which uniquely determines an explicit XML node, thus structure sharing on the KST signature can be decompressed.

4. Encoding of pointers

We propose separating the pointer implementation for sharing of compressed XML data into two parts: the part encoding the origin, and the part encoding the destination. While the encoding of the pointer origin is described in the first sub-section, the encoding of the pointer destination is described the second sub-section.

4.1. Encoding of pointer origins

Pointers to shared structures could be encoded externally in a pointer table storing origin and destination of pointers. Alternatively, pointers can be encoded inline in the KST signature. As a pointer origin may occur after each number or bit of the KST signature, inline encoding requires distinguishing between pointers and other KST signature tokens. We use a special escape bit-sequence to describe that a pointer follows in the KST signature. The detection of this escape sequence can be done very efficiently by using an automaton. Our comparative computation of pointer sizes has shown that for pointers within KST signatures, inline encoding compresses much better than external encoding of pointers.
4.2. Encoding the pointer destination

As all pointers within the DAG point backwards to already known sub-trees, we decided to encode the address of the pointer destination in terms of the 'distance' from the pointer origin to the pointer destination. As a metric for the distance, we use the number of SAX events between the pointer destination, which is the first occurrence, and the pointer origin, which is the repeated occurrence of the common sub-tree within the XML document. This metric is much more compressible than pointers addressing bits or bytes in the KST signature.

4.3. Example

The compression result depends on the DTD. If we compress the XML document shown in Table 1 (b) using the following DTD which differs in the element definition of the element 'e' from that of Table 1 (a):

```
<!ELEMENT e (f?, f?)>
<!ELEMENT f (g, h, i?)>
<!ELEMENT g (EMPTY)>
<!ELEMENT h (EMPTY)>
<!ELEMENT i (EMPTY)>
```

we get a different list of explicit nodes, and a different resulting KST signature:

```
'f  i  f  i'
'1  1  1  1'
```

where each '1' within the KST signature denotes the existence of an 'f'-element or of an 'i'-element. If we assume, that the special token for escaping pointers is 00 and we wanted to implement the KST signature pointer for the sub-tree behind the second 'f' element, instead of encoding this sub-tree in lines (8)-(10) of the XML document, the resulting KST signature would be

```
'1 1 1 00 7'
```

where 7 is the distance from the SAX event of line (8) to the SAX event of line (3) (as e.g. '<h/>' corresponds to both one startElement and one endElement SAX event).

4.4. Avoiding losses in compression ratio

As can be seen from the previous example, implementing a pointer might lead to a longer encoding than repeating shared sub-trees. Therefore, we only use pointers when the encoding using pointers is shorter than the repetition of the shared sub-tree.

Therefore, whenever our compression finds a sharable sub-structure of the KST signature, it simultaneously calculates the encoding of the pointer and the encoding of the shared sub-tree. Afterwards, it writes the shorter of both encodings into the KST signature. Thereby, we can ensure that the combination of KST signature and structure sharing results in a compressed representation that is not larger than the KST signature on its own.

This avoidance of losses in compression ratios implies especially, that too small sharable sub-trees are not encoded by a backward pointer. As the number of shared sub-trees really saving space is smaller for KST signatures than it is for raw XML documents, applying sub-tree sharing to the KST signature will not result in the same compression increase as applying sub-tree sharing to the raw XML data.

To summarize, the improvement of KST signature over DTD subtraction can be observed especially for XML documents with large repetitions of common sub-trees (because of the backward pointers) and for XML document with a large number of siblings of the same element, represented in the DTD in form of a '+'- or a '*'-operator (because of the improved encoding of these operators).

5. Query processing

When we consider XPath query processing on KST signatures, we have to evaluate the XPath expression itself, and we may have to return the decompressed XML sub-trees of the nodes matching the query.

Query evaluation on the shared KST signature is based on the following key ideas and concepts.

- We normalize queries in order to minimize set of operations to be implemented and in order to simplify the presentation.
- We use a stack of entries representing XML element nodes on the child path from the root node to the current context node. Each stack entry contains the information necessary to continue query processing.
  - More precisely, each stack entry contains the element's name, a text constant counter, a SAX event counter, the current KST signature position, and the current parsing position in the DTD rule defining the element's parent.

We regard query processing for the following subset of XPath:

```
cxp ::= '/' locationpath
locationpath ::= locationstep ('/' locationstep)*
locationstep ::= x '::' t | x '::' t '[ ' pred ' ]'
pred ::= pred 'and' pred | pred 'or' pred | 'not' '(' pred ')' | locationpath | locationpath '=' const | ‘(‘ pred ‘)’
```

Proceedings of the 42nd Hawaii International Conference on System Sciences - 2009
“cexp” is the start production, “x” represents an axis (attribute, self, child, parent, descendant-or-self, descendant, ancestor-or-self, ancestor, following, preceding, following-sibling, preceding-sibling), “const” represents a constant, and “t” represents a “node test” (either an XML node name test or “*”, meaning “any node name”).

5.1. Query normalization

In order to simplify the presentation and to reduce the number of location steps to be implemented, we treat attributes like child elements and we normalize queries, such that they contain only the basic axes first-child(FC), next-sibling(NS), parent (P) and label(L).

In the first step, we substitute each location step following::x with the equivalent location step sequence ancestor-or-self::*/following-sibling::*/ descendant-or-self::x , and we replace each occurrence of preceding::x with ancestor-or-self::*/preceding-sibling::*/ descendant-or-self::x .

In a second step, we eliminate backward axes by using the “looking forward approach” [13], which rewrites each XPath query using backward axes into an equivalent XPath query using forward axes only.

In a third step, we further reduce arbitrary XPath location steps to the basic steps first-child, next-sibling, label, parent as motivated in [10].

After the query normalization, the queries contain only the axes first-child, next-sibling, parent and label.

5.2. Supporting the basic steps on DAG KST

An FC step is executed as follows. The first-child ‘fc’ of an element ‘p’ is that element, the start tag of which follows directly after the start tag of ‘p’. Therefore, we parse the KST signature using the DTD. If parsing reaches the end of the regular expression ‘r’ defining the element ‘p’, before a first child element is found, then no first child exists. Otherwise, the first child element of ‘p’ generated as a side-effect of parsing the KST with the DTD is returned as the first child of ‘p’.

For example, if we look for the first child of the node ‘e’ in the example of Section 4.3., the first ‘1’ in the KST signature tells us that there is a child ‘f’ of ‘e’ which is returned as the first child.

A NS step is executed as follows. The next-sibling ‘ns’ of a current context element ‘ps’ is that element, the start tag of which follows directly after the end tag of ‘ps’. Therefore we first have to find the end tag of ‘ps’. In order to find the end tag of ‘ps’, we parse the KST signature using the DTD until we reach the end of the DTD rule defining ‘ps’. (In case of recursive DTDs parsing continues until we reach the end of the same invocation of the DTD rule for ‘ps’.) While parsing through DTD rules and KST signature, we have to count PCDATA-Elements within the DTD in order to provide direct access to compressed text values. After we have reached the end tag in this manner, the next-sibling will be the next element returned by continuing the parsing of the DTD rule defining the parent of ‘ps’.

If parsing reaches the end of this rule without returning an element, no next-sibling exists.

For example, if we look for the next-sibling of the first ‘f’ in Example 4.3. which was represented by the first ‘1’ in the KST signature, parsing continues until the end of the DTD rule defining ‘f’ and consumes the second ‘1’ bit of the KST signature representing the existence of the ‘i’ child node of the first ‘f’ element. Then parsing continues with the DTD rule defining ‘e’, which is the parent of ‘f’ behind the first occurrence of ‘f’? which just has been processed. As the element declaration for ‘e’ contains a second entry ‘?’ the next KST position, i.e. the third ‘1’, is evaluated to determine that a next sibling ‘f’ exists.

In order to support fast navigation along the P axis, we keep a stack of previous query evaluation states which contains one entry for each element node on the child path from the XML document root to the current context node. Each stack entry contains the information required to continue query processing at the element node represented by the stack entry, including the element node’s SAX event counter, text event counter, current position in the KST signature, and current position in a DTD rule.

A navigation to the FC yields a new entry on the stack, whereas a navigation to the NS results in popping the top-most entry from stack and adding a new entry to the stack.

When a P step is executed, we pop the top-most entry of the stack.

5.3. Query evaluation on shared KST signatures

The stack entries support query processing on KST signatures using pointers as follows. When query evaluation requires using a pointer to a common sub-tree, the pointer destination’s SAX event number is used for finding the first stack entry with a less or equal SAX event number, i.e. the first stack entry SE of an element being or enclosing the pointer destination. If the SAX event numbers of the pointer destination and of SE are the same, the element referred to by the pointer is found, and the stack entry SE contains all the information to continue query processing, such as KST signature position and DTD rule position. Otherwise, SE describes an ancestor element of the pointer destination, and the pointer destination’s SAX event number is used as an index for
searching the pointer destination among the descendants of the element represented by SE.

Sharing of common sub-structures by using pointers requires returning to the calling place and continuing further processing from there, when a sub-structure has been fully processed. Therefore, for each pointer to a shared substructure that is used during query evaluation, we generate a stack entry containing the continuation information needed after the shared substructure is processed, which includes pointer origin, KST position and DTD rule position. This stack entry is popped after returning from a shared substructure.

5.4. Answer tree generation and decompression

As we see in Section 6.2, queries can be evaluated on the KST signature nearly as fast as on the original XML representation. Therefore, in most cases, a decompression of the KST signature is not needed. For the remaining cases where decompression of KST substructures is needed, which is e.g. the case for the generation of decompressed XML answer sub-trees to a query, the document can be partially decompressed as follows.

Concerning the raw KST signature without any pointers for structure sharing, the decompression simply performs the inverse actions to the compression, and it can be implemented by using the DTD to parse the KST signature. Whenever the decompressor reads a pointer referring to a previous SAX event, it follows the pointer as this is done for queries and continues decompression at the node referenced by the pointer.

6. Performance evaluation

6.1. Evaluation of compression ratios

We have implemented KST signature using Java 1.5 and a SAX parser for parsing XML documents. We have evaluated KST signature on the following datasets:

- XMark (XM – 5.3 MB) – an XML document that models auctions [14]
- hamlet (H – 0.3 MB) – an XML version of the famous Shakespeare play
- catalog-01 (C1 – 10.6 MB), catalog-02 (C2 – 105.3 MB), dictionary-01 (D1 – 10.8 MB), dictionary-02 (D2 – 106.4 MB) – XML documents that were generated by the XBench benchmark [17]
- dblp (DB – 308.2 MB) – a bibliographic collection of publications

We compared KST signature with six other approaches:

- XGrind [15] – a queryable XML compressor,
- gzip – a generic compressor based on Huffman encoding and LZ77,
- BZip2 - a generic compressor based on Burrows-Wheeler-Transformation,
- DTD subtraction [3] – a structure-based XML compressor without optimized signature encoding and without structure sharing,
- BSBC [2] – a queryable XML compressor combining XML encoding with DAG-based structure sharing and using BZip2 for the compression of constant values,

The results of our experiments are shown in Figure 1, where a compression ratio of 20% means that the size of the compressed file is only 20% of the size of the original uncompressed file. Using these datasets, KST signature performs better than each of the compared compressors BZip2, gzip, XGrind, DTD subtraction, BSBC and XMill for all documents except for hamlet – the smallest document within our evaluation – where KST signature is outperformed only by gzip and BZip2, which both produce non-queryable data.

![Figure 1. Compression ratio of the whole XML document](image-url)
In order to show the improvement of KST signature compared to DTD subtraction, we have measured the structure compression ratio, i.e. (size of KST signature) / (size of XML structure), in a second series of measurements. As shown in Figure 2, we have achieved a significant improvement yielding an improvement factor for the compressed document structure between 3.5 and 7.

6. Evaluation of compression and decompression times

Figure 3 summarizes our evaluation results on the data throughput of the compression techniques. gzip yields the highest throughput reaching rates of up to 12000 bytes/ms, XMill is the second fastest followed by Bzip2 and finally followed by the queryable compressors DTDSUB, BSBC and KST signature.

However, Figure 4 shows that most of the compression time of KST signature compression, i.e. between 70% and 90%, is consumed by BZip2 for compressing the data values, whereas our KST signature approach uses only 10-30% of the time to compress the XML structure.

We got very similar evaluation results, shown in Figures 5 and 6, for the decompression throughput and the decompression times, with the only difference that decompression is faster than compression for all approaches.
6.3. Evaluation of query performance on compressed data

In order to evaluate the query performance on compressed data, we have compared the query evaluation on KST signature with two other approaches. On the one hand, we have compared it with JAXP, the standard XPath evaluator contained in Java 1.5 in order to compare the overall query performance. On the other hand, we have compared it with our generic query evaluation engine working on uncompressed SAX events and using the same elementary navigation operations first-child and next-sibling and the same framework on top to evaluate queries, in order to evaluate the effect of compression on the query evaluation time.

Our test data set was generated by the XML generator of the XML Benchmark XMark [14] using a scaling factor of 0.001 (116 kB) up to 0.128 (14.5 MB).

On our dataset, we have evaluated queries of the XPath benchmark XPathMark-A[8]. The test queries can be seen in Table 2.

<table>
<thead>
<tr>
<th>ID</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>/site/closed_auctions/closed_auction/annotation/description/text/keyword</td>
</tr>
<tr>
<td>Q2</td>
<td>/site/closed_auctions/closed_auction//keyword</td>
</tr>
<tr>
<td>Q3</td>
<td>/site/closed_auctions/closed_auction/annotation/description/text/keyword/date</td>
</tr>
<tr>
<td>Q4</td>
<td>/site/closed_auctions/closed_auction/descendant::keyword/date</td>
</tr>
<tr>
<td>Q5</td>
<td>/site/people/person/profile/gender and profile/age/name</td>
</tr>
<tr>
<td>Q6</td>
<td>/site/people/person/phone or homepage/name</td>
</tr>
<tr>
<td>Q7</td>
<td>/site/people/person/address and (phone or homepage) and (creditcard or profile)/name</td>
</tr>
</tbody>
</table>

Figure 7 shows the test results of all queries for the largest document with scaling factor 0.128 and a size of 14.5 MB. Except for the queries Q2 and Q4, KST signature performs better than JAXP and performs comparable to our generic query evaluation engine working on the uncompressed XML data. KST signature needs a longer evaluation time than the other approaches for the queries Q2 and Q4, as the descendant-axis location step in these queries requires that the whole sub-tree under the current context node has to be searched and thus partially decompressed. As KST signature was designed for optimal compression, an index for descendants has not been included.

Figures 8 and 9 show the scaling of queries Q2 (where KST signature performs worse than the other two evaluators) and Q7 (where KST signature performs best of all three evaluators). As can be seen in both figures, KST signature’s query evaluation time scales linearly depending on the size of the uncompressed XML document. Therefore, it can be used for extremely large documents as well.
7. Summary and conclusions

We have presented a technique for lossless data reduction of huge XML files and XML data streams. Our data reduction technique combines KST signature compression with structure sharing known from DAGs in the following way.

We introduce the concept of explicit XML nodes that can be used as pointer origin and still allow for unique decompression and fast query evaluation. Furthermore, we encode pointer destinations by SAX event numbers of previous XML nodes.

We have integrated the concept of pointer to substructures with the KST signature that can be generated by parsing the XML document with an attribute grammar that uses the DTD. A special feature of the compression and the attribute grammar rule is that the same DTD rule set can be used for decompressing the KST signature. Query processing avoids full decompression and uses partial decompression on demand, e.g. for answer sub-tree generation.

We have reduced query processing to a set of elementary navigation operations, consisting of first-child, next-sibling, parent and label, and we have developed fast query evaluation on top of these elementary operations. To support following pointers and backward navigation, we keep the child path from the root to the current context node on further information on a stack. This supports efficient query evaluation on our compressed data format KST signature without the requirement of expensive decompression.

Finally, we have conducted extensive performance evaluations that show that our approach not only results in reasonable query evaluation times, but also outperforms the compression ratios of all the other compression techniques, including even non-queryable compression techniques like gzip, XMill and BZip2.

Altogether, our approach reduces huge XML data structures or XML data streams to a significantly smaller KST signature. It combines a scalable and significant size reduction of verbose XML structures with a comparably fast query processing technique. Therefore, the use of KST signature is a significant step towards XML data reduction and the management of huge XML files and streams.

8. References