

CSC: Supporting Queries on Compressed Cached XML

Stefan Böttcher, Rita Hartel

University of Paderborn,
Computer Science, Fürstenallee 11,
33102 Paderborn, Germany
+49 52 51 60 66 62, +49 5251 60 66 12

stb@uni-paderborn.de, rst@uni-paderborn.de

Abstract

Whenever a client frequently has to retrieve, to query and to locally transform large parts of a huge XML document that is stored on a remote web information server, data exchange from the server to the client may become a serious bottleneck that simply limits scaling of the amount of information that can be processed locally on the client by a client-based application.

We present Compressed Structure Caching (CSC) as a solution that reduces the amount of data exchange by a combination of the following techniques: compression of the XML document's structure, client-side caching of the structure and of already received XML content, inference and optimized loading of the content needed on the client to answer a given query.

We provide a performance evaluation that demonstrates that our approach significantly reduces the amount of data exchange from server to client.

Keywords: XML, Caching, Compression.

1 Introduction

1.1 Motivation

XML has become a standard data exchange format in many information sources e.g. in the web, and XPath has become a key standard for context sensitive search in huge XML documents.

We consider scenarios, in which client applications need to process large fragments of huge XML documents that are provided on remote web servers, and where the data exchange from the server to the client is a bottleneck. These scenarios require techniques that minimize data transfer between the server-side XML information source and the client submitting queries. In order to reduce the data transfer, two techniques are possible: compression of exchanged data, and caching and reuse of previous query results. Both techniques have been investigated independently of each other, but are challenging to combine.

Our approach combines both techniques, i.e., it is based on caching and reusing compressed XML information which the client has previously downloaded from the server. The problem considered in this paper is how to reduce the amount of XML data exchange between server and client by intelligent server-side XML fragment compression and by a client-side caching and integration strategy for compressed XML data, such that XPath queries can be executed on cached compressed data on the client.

1.2 Limitations of related approaches

Related approaches follow two different directions called query shipping and data shipping. *Query shipping* means that each client C that cannot answer a given query Q_i from its cache sends Q_i to the information server that executes Q_i on the information source IS, i.e. computes $R=Q_i(IS)$ and returns the result R to C. The returned result R is stored in the cache and can be used to answer a second query Q_k if a compensation query Q_c exists such that Q_k applied to the information source IS returns the same answer as applying Q_c to R. More formally, the returned result R can be used to answer a second query Q_k if it can be proved that $Q_k(IS) = Q_c(R)$. The proof techniques suggested by e.g. (Balmin et al. 2004), (Mandhani and Suciu 2005), (Xu and Ozsoyoglu 2005) aim at showing that Q_k is equivalent to $Q_c \circ Q_i$, which is sufficient to prove that $Q_k(IS) = Q_c(Q_i(IS)) = Q_c(R)$.

Unfortunately, the compensation query approaches are applicable to very small subsets of XPath only. Even worse, it can be shown that already for very small subsets of XPath the search for compensation queries is NP hard.

As a consequence most queries can not profit from the cache when using query shipping and the same XML fragments may be shipped as a part of an answer again and again.

Data shipping means that all the data needed to answer a query Q_i is shipped to the client such that Q_i can be answered locally. For example, approaches like (Böttcher and Türling 2004), (Koch, Scherzinger and Schmidt 2008), and (Marian and Siméon 2003) have been developed to compute the so called *read set* of a query, which is an easily computable superset of the data that has to be accessed to answer the query.

For the purpose of reducing data transfer, the data shipping approach has the following advantages: A client can determine very fast which part of the data needed to answer the query can be read from its cache, and in the long run, the data shipping approach to query processing

increases the number of possible cache hits. Although data shipping ensures a greater amount of cache-hits than query shipping, a huge amount of data must be transferred for queries the read-set of which is very large, e.g. queries that count data or queries that involve a long search on the XML structure. In other words, the disadvantage of data shipping is an increase of transferred data that is not needed to answer a query.

Regarding the advantages and disadvantages of query shipping and data shipping, our goal is to combine the advantages of both query processing approaches. Although there is a trade-off between additional data transfer and number of cache-hits, we show that our approach results in a significant reduction of overall data transfer between the web-information source and the client.

1.3 Problem description

The problem investigated is how to improve web information caching in such a way that the overall data exchange needed between the web information source and the client is reduced for arbitrary XPath queries. This includes XPath queries which have to search or to navigate in a large part of the structure of an XML document and XPath queries that use axes beyond the limited sub-classes that have been investigated for compensation queries.

1.4 Contributions

This paper proposes a novel approach to caching of XML web information, called Compressed Structure Caching (CSC) that combines the following properties:

1. It separates a huge XML document that is provided on a web information server into its constituent parts: its tree structure and its values of text constants and of attributes.
2. It extracts and compresses the structure of the XML document to a compressed tree (CT) including the element names and attribute names and transfers the CT to the web clients that want to work on the XML document.
3. The client can decide based on the CT whether or not he has enough knowledge to answer the query on its own, i.e., without sending the query to the server and retrieving the query results from the server.
4. Web clients submit queries to the server which infers from each client query which text and attribute values are required on the client to answer the query. The required values are compressed and sent to the server.
5. Finally, the client caches the CT plus the compressed values and evaluates its queries on these compressed structures.

We have implemented and comparatively evaluated our system (CSC) with two other approaches, i.e. querying uncompressed data, and querying compressed data that is not kept in the cache. Our results show that SCS clearly outperforms the two other approaches.

1.5 Paper organization

The remainder of this paper is organized as follows. In Section 2, we explain the key ideas of the general solution and show the system architecture. Section 3 describes a specific solution instance for which we have done the performance evaluation and the evaluation results. Section 4 describes related work and Section 5 contains a summary and the conclusions.

2 Key ideas of the CSC solution

We first give an overview of the system architecture and thereafter describe the key ideas and design decisions of our implementation of CSC.

2.1 System architecture

The overall architecture of our CSC system is shown in Figure 1.

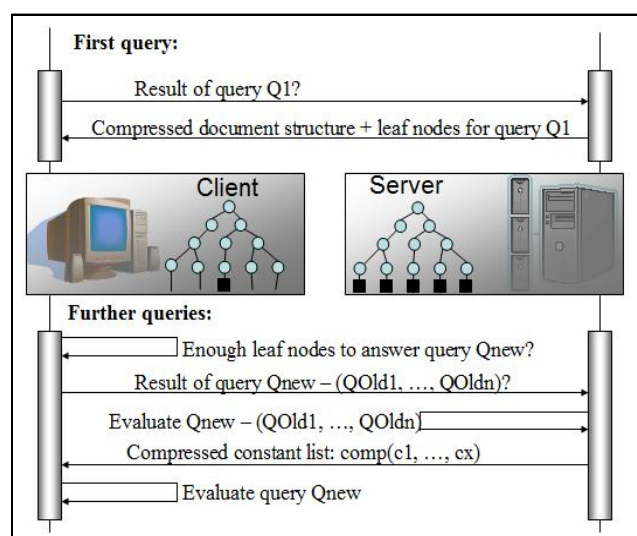


Figure 1. Overview of the system architecture

The server that provides the huge XML information source separates the XML tree structure from the values of the XML leaf nodes, i.e. texts and attributes, and compresses the XML structure without the XML leaf node values separately.

When a client submits its first query to the server, the server transfers the highly compressed XML structure to the client. Additionally, for each query, the server collects all the XML document's leaf nodes in document order that are needed for query evaluation, but that are not contained in the client's cache. The server compresses this XML leaf node constant list and sends it to the client. Finally, the client evaluates the query.

Before the client sends further queries to the server, it checks whether the query can be answered locally or requires more XML leaf node data. Only if data is missing, in the client's cache, the client submits the query to the server.

2.2 Key idea of CSC

The key idea of the implementation is that client and server use the same XPath evaluator EV with one exception: the implementation of the access to constant values. The main requirement to EV is that the program code accessing constant values is isolated, e.g. it is done

in a function *getValue(XPathQuery, currentContextNode)* for which a server implementation exists that differs from the client implementation. Client implementation and server implementation communicate via the constant exchange buffer explained below. All other operations of the XPath evaluation EV are implemented only once, and are used in an identical fashion on the server side and on the client side.

Client and server use the same XPath evaluator, in our case an evaluator based on a reduced instruction set RIS. RIS consists of the following operations:

- *fc*: Returns the first-child of the current context node *ccn*.
- *ns*: Returns the next-sibling of the current context node *ccn*.
- *label*: Returns the label of the current context node *ccn* if *ccn* is an element or an attribute node.
- *parent*: Returns the parent of the current context node *ccn*.
- *node type*: Returns the node type (i.e., either element, attribute, or text node) of the current context node *ccn*.
- *getValue*: Returns the text value of the current context node *ccn* if *ccn* represents a text node or the text value of an attribute.

Other operations of CoreXPath are reduced to these operations using rewrite rules as presented in (Böttcher and Steinmetz 2007a).

The main goal of the constant exchange is to send only those constants from the server to the client that are really needed. This will save most of the data exchange, because the constants are much more difficult to compress than the structure.

The key idea is to use the same XPath evaluator on the server and on the client side. The server's evaluator only picks those constants from the document that are needed on the client side. These constants are packed together, are compressed using string compression (e.g. in our case *bzip2*), and finally are submitted to the client and decompressed. As the evaluation order of XML nodes on the client side and on the server side are identical, the constant order picked by the server is identical to the constant order required by the client. Therefore, the server simply writes picked constants sequentially into the buffer that is compressed, submitted to the client and decompressed at the client side. And the client simply reads the constants from the decompressed buffer one by one.

2.3 Separation of structure and text constants

Due to the semi-structured nature of an XML document, the document structure contains a lot of redundancies, while the text data does contain fewer redundancies. Therefore, the document structure alone can be compressed much better than text and attribute values alone or the XML document as a whole combining structure and constants. For example, the overall compression ratio achieved for different XML documents including structure plus text data reaches compression ratios of 10% up to 30%, i.e., it reduces the document size by a factor of 3.3 up to 10. However, the compression

ratio achieved for structure only of the same XML documents is between 0.3% and 10%, i.e., it reduces the document structure size by a factor of 10 up to 330. Furthermore, nearly all XPath queries access significantly more inner XML document nodes which are part of the XML structure than they access leaf nodes which contain the text or attribute values.

Therefore, we propose to separate the structure from the text constants and compress both parts separately. Compression approaches that perform such a separation of structure and constants and that support queries and even modification on the generated compressed XML at the same time are e.g., BSBC (Böttcher, Hartel and Heinzemann 2008) and DTD subtraction (Böttcher, Steinmetz and Klein 2007). In general, any compressor that separates structure from text constants and that support queries and modification on the generated compressed XML can be integrated in our approach to Compressed Structure Caching (CSC) as well.

2.4 Caching the complete structure, but constants only on demand

Due to the strong compression achievable for structure, it is much more likely that the complete structure can be kept in the cache than that the complete XML document or huge fragments of it can be stored in cache. Therefore, one of the key ideas of CSC is to keep the complete compressed document structure within the client's cache.

However, the constants are loaded into the client's cache on demand, i.e., only when they are needed in order to answer a query. The benefit of this idea is that the inner XML document nodes, i.e. the huge majority of nodes needed for query processing is already available in the cache, whereas only a small minority of nodes, i.e. the leaf nodes really accessed, have to be loaded if not already present in cache.

As the complete document structure and a subset of the text constants are known on the client, the XPath evaluation can be started at the client side in order to determine, whether or not text constants or attribute values are missing. That means that the client can decide based on its XML cache, whether or not it contains already all the data needed to answer the queries.

In contrast, other approaches like the proof techniques suggested by e.g. (Balmin et al. 2004), (Mandhani and Suciú 2005), (Xu and Ozsoyoglu 2005) aim at showing that a so called compensation query applied to the cache returns the same results as the original query applied to the server's database.

Unfortunately, the compensation query approaches are applicable to very small subsets of XPath only and even worse, it can be shown that already for very small subsets of XPath the search for compensation queries is NP hard.

2.5 Pointer-less identification of constants

Pointers from the XML structure into a compressed text or attribute value constant buffer may speed-up query processing, but the addition of pointers to the compressed structure of an XML document will significantly blowup size of the compressed structure, usually by more than 100%. In comparison, a pointer-less technique to address

the relevant constants significantly reduces the space needed for the compressed XML structure.

Therefore, CSC uses a constant identification technique that avoids the need for pointers from the compressed structure to the compressed constant values. This constant identification technique saves cache space to store larger XML structures and thus reduces the data transfer from the server to the client.

The key idea used by CSC is the following.

As the XPath evaluation can be started on the document structure, the evaluation order gives an implicit order of the needed text constants. When the server uses the same evaluation order as the client, the server can send compressed text constants in this evaluation order, and the client receives the constant in the evaluation order needed. Therefore no explicit pointers or identification of the constants is needed - neither to transfer constants, nor to insert the constants in the cache, nor to use the constants on the client side. This pointer-less access to the needed constants results in a significant reduction of transferred data.

Note that other caching techniques, e.g. (Böttcher and Türling 2004) use an additional addressing or identification schema for XML nodes, as e.g. the ORDPATH numbering scheme (O'Neil et al. 2004), in order to integrate additional data from the server into the client's cache. This address information requires cache memory and has to be transferred, both being avoided by our CSC approach.

2.6 Constant identification on the server side

On the server side, a modified XPath evaluator not only evaluates the query, but simultaneously collects all the constants of accessed XML leaf nodes in evaluation order, which in our special case is XML document order. From this collection of accessed XML leaf node constants, all the constants that are already known to the client's cache are deleted during the same scan through the XML document by a technique described in Section 2.7. Thereby, the result of this server side query evaluation is a list of constant values of accessed XML leaf elements in evaluation order, except the values of those leaf elements that are already stored in the client's cache. This list of constants is transferred to the client without any additional identification information, as this information is implicitly known to the client because of the document structure and the evaluation order.

2.7 Constant usage on the client side

The server and the client have to agree on a common XPath evaluator to ensure that the evaluation order of constants is the same on server and on client.

Before sending the query to the client, the client tries to evaluate the query on its cache. During evaluation it will either realize, that no constant data is missing. In this case, no data has to be transferred between server and client. If on the other hand, the client realizes, that its cache does not store all the leaf node values needed to answer a query independently of the server, the client sends the XPath query string to the server. There it is evaluated by a modified XPath evaluator that collects

only those values of texts and attributes that are accessed, but not contained in the client's cache in order to answer the query. The values collected by the XPath evaluator are compressed and sent to the client. After having received the list of constants, the client can continue the query evaluation. Whenever a constant value is missing, the client consumes the next constant value from the list, stores it on the current position within the compressed document and uses it for client evaluation. As client and server have agreed on the same evaluation order, this ensures, that the next constant value received from the server is the next constant value the client needs.

This allows a compression and decompression technique of string constants where the server simply pipes in strings into the compressed stream to the client and the client simply extracts them one by one.

2.8 How to avoid the transfer of leaf nodes that are stored in the client's cache

In order to prevent the server from sending constants to the client that are already stored in the client's cache, either the server has to know, which constants are stored within the client's cache, or the client has to tell the server which constants are still stored in the cache.

To avoid sending the same XML leaf constants several times, we have adopted and slightly modified an approach of (Böttcher and Türling 2004). Each query string submitted from the client to the server is stored in a client-specific query list on the server. The server compares this query list with the IDs of old queries to compute the list of *cached queries*, i.e. those queries, the results of which are still stored in the client's cache. The list of cached queries and the actual query are combined to compute the missing constants. Here *missing constants* denote the constants needed to answer the actual query on the client-side that are not yet contained in the cache, i.e. which are the cache-misses and have to be sent from the server to the client. To avoid reading the XML source multiple times, we use a streaming-based approach to read the whole XML document in a single pass only based on (Olteanu et al. 2002) and (Böttcher and Steinmetz 2007a), and apply multiple queries, i.e. the actual query and the cached queries in parallel on this stream. This provides an easy way to compute the leaf nodes accessed by the actual queries and the leaf nodes accessed by the cached previous queries in a single run on the compressed XML file. Only those leaf nodes that are not yet present in the client's cache are collected in document order, are compressed, and are then sent to the client.

2.9 How the client embeds the received constants

CSC has to adapt its pointer-less identification of constants to the situation that some, but not all constants are already present in the client's cache. This means that for each leaf node visited by the client's XPath evaluator, the evaluator has to know whether the constant is stored in the cache or is contained in the stream of constants retrieved from the server.

This can be achieved by using one bit for each leaf node in the cached structure telling the XPath evaluator whether the constant has to be read from the client's cache or from the stream of constants provided by the server.

3 Performance evaluation

Our performance evaluation was done with BSBC (Böttcher, Hartel and Heinzemann 2008) as structure compression tool and Bzip2 as text compressor.

3.1 Summary of BSBC

In our performance evaluation, we have used BSBC on both, the client and the server, as the compressor that generates queryable and updateable compressed XML data and as the tool that performs queries on BSBC compressed XML data.

BSBC is an XML compressor based on element name encoding on the one hand, and on sharing of common sub-trees on the other hand. The BSBC XML compressor separates the XML constants from the XML element names and attribute names and from the nesting of start tags and end tags, i.e. the compressed document structure of an XML document consists of the following parts:

- i. A bit stream representing the tree structure of the element nesting in the XML tree, without storing any label information. In the bit-stream, each start-tag is represented by a '1'-bit and each end-tag is represented by a '0'-bit.
- ii. Inverted element lists, containing a mapping of element and attribute names to '1'-bit positions within the bit stream.
- iii. A so called DAG pointer list. The DAG pointer list represents the shared sub-trees, and it consists of a list of pointers from a parent element to the repeated sub-tree occurring previously within the document structure.

In addition to the document structure, BSBC stores the constants, i.e., the text and attribute values in form of separate constant containers based on the parent element name and compresses each constant container using the generic compressor BZip2.

3.2 Performance evaluation environment

We have implemented Compressed Structure Caching using Java 1.5. We have evaluated Compressed Structure Caching on a dataset created by the XMark Benchmark (Schmidt et al. 2002) using creation factors from 0.0001 to 0.1 and yielding document sizes from 34 kB (factor 0.0001) up to 11.3 MB (factor 0.1).

We compared three different models

- CSC: Compressed Structure Caching described in this paper that initially transfers the whole structure compressed by BSBC. For each query, only the missing constants of XML leaf nodes are transferred, and they are transferred in a compressed way.
- Compression: The query results are computed on the server and the results are only compressed by BSBC

before transferring them to the client, i.e. caching is not used.

- Direct: The query results are computed on the server and are transferred to the client in an uncompressed way.

3.3 Performance results

In order to compare the three models, we have executed the queries shown in Table 1 sequentially and have measured the total data volume transferred from server to the client.

ID	Query
q0	/site/people/person[phone or homepage]/name
q1	/site/people/person[descendant::watches]
q2	/site/regions/europe/item/name
q3	/site/people/person/address/city
q4	/site/open_auctions/open_auction/bidder
q5	/site/closed_auctions
q6	/site/people
q7	/site/open_auctions
q8	/site/categories
q9	/site
q10	/site/regions/europe

Table 1. Queries used to evaluate the proposed approach

Figures 2 and 3 show the transferred data volume for the documents created with factors 0.0001 and 0.1. A comparison of both figures demonstrates that after a certain amount of queries, Compressed Structure Caching (CSC) transfers less data than the compressed model that transfers itself less data than the direct model. The bigger the document on the server is, the earlier this effect can be realized: For the document XMark 0.0001, the structure cache transfers less data than the compressed model only from query q9 on, whereas it transfers less data from query q6 up for document XMark 0.1.

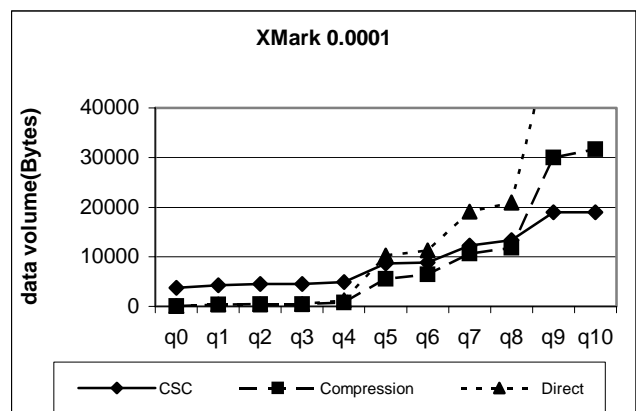


Figure 2. Transferred data volume for XMark 0.0001

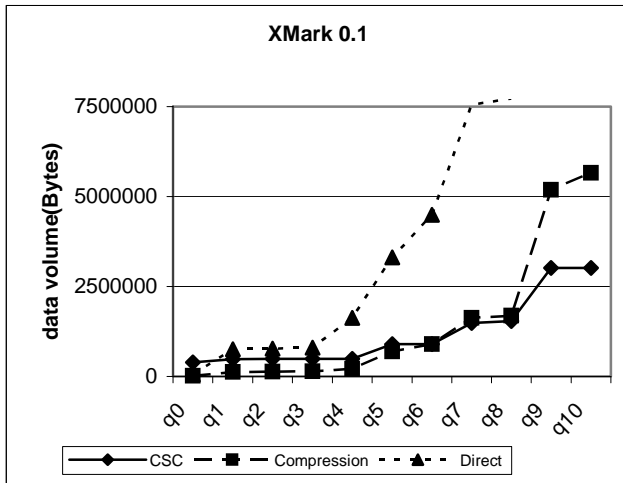


Figure 3. Transferred data volume for XMark 0.1

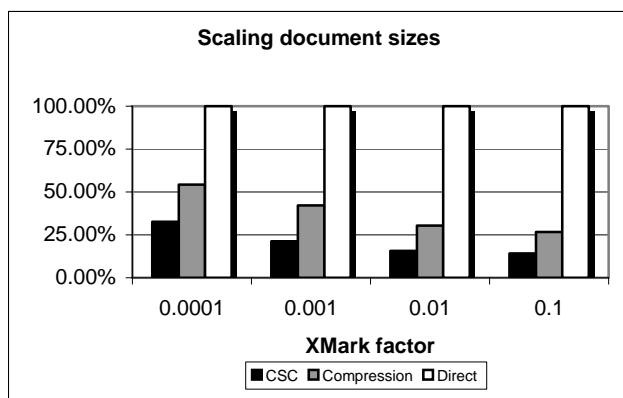


Figure 4. Transferred data volume for different document sizes

This is mainly due to the relative size of the compressed structure which is initially transferred from server to client, and which is 10% of the document size for XMark 0.0001, but only 3% of the document size for XMark 0.1.

We can take a closer look on this effect in Figure 4: When using the compressed model without caching, the data volume that has to be exchanged can be reduced to the size of 55% to 26%, compared to the data volume exchanged with the Direct strategy using neither caching nor compression. The reduction of the exchanged data volume is better for larger XML documents.

When using the compressed model with caching, the data volume that has to be exchanged can be reduced even significantly better, i.e. to the size of 33% to 14%, compared to the data volume exchanged with the Direct strategy using neither caching nor compression. Again, the reduction of the exchanged data volume is better for larger XML documents.

To summarize, it can be seen, that even for relative small document sizes and few queries, by using Compressed Structure Caching, the data volume can be reduced up to 13% compared to the direct model and up to 50% compared to using the compressed model.

4 Related work

Although both, web data caching and XML compression, contribute to a reduction of the data transfer from server to client, the fields of web data caching and XML compression have mostly been investigated independently of each other.

There has been a lot of work in XML compression, some of which does not support query processing on compressed data, e.g. (Liefke and Suciu 2000), and most of which support querying compressed data, but not querying cached compressed data, e.g. (Buneman, Grohe and Koch 2003), (Busatto, Lohrey and Maneth 2005), (Cheng and Ng 2004), (Ng et al. 2006), (Zhang, Kacholia and Özsu 2004).

Contributions to the field of caching range from concepts of querying and maintaining incomplete data, e.g. (Abiteboul, Segourin and Vianu 2001), over caching strategies for mobile web clients, e.g. (Böttcher and Türling 2004), to caching strategies based on frequently accessed tree patterns, e.g. (Yang, Lee and Hsu 2003). In comparison, our approach allows for XPath queries using filters and comparisons with constants even on compressed cached XML.

Different approaches have been suggested for checking whether an XML cache can be used for answering an XPath query. On the one hand, there are contributions, e.g. (Balmin et al. 2004), (Mandhani and Suciu 2005), (Xu and Ozsoyoglu 2005), that propose to compute a compensation query. These approaches can also be used on compressed XML data, but they are NP-hard already for very small sub-classes of XPath. On the other hand, containment tests and intersection tests for tree pattern queries have been proposed, and could in principle be used for deciding whether a given XPath query can be executed on the cached data locally. However, such intersection tests and containment test are NP-hard for rather small subsets of XPath expressions (Benedikt, Fan and Geerts 2005), (Hidders 2003). In comparison, our approach uses a fast difference computation that can be done within a single scan through the compressed XML file.

In comparison to all other approaches, our technique is to the best of our knowledge the only strategy that combines the following advantages: it caches the relatively small compressed XML structure and supports XPath queries on it, transfers only constants that are really needed for query evaluation and uses a pointer-less transfer format, and it uses an intelligent server strategy to identify those leaf nodes not yet stored in the client's cache.

5 Summary and Conclusions

Whenever data exchange with XML-based information sources is a bottleneck, it is important to reduce the amount of exchanged XML data. Our CSC approach combines two reduction techniques for exchanged data, i.e. caching and XML compression. Furthermore, CSC supports XPath query evaluation on cached compressed XML data, and is not limited to a small XPath subset like tree pattern queries. Additionally, CSC takes advantage of caching the small compressed XML structure and

provides an intelligent technique for transferring only those XML leaf node constants from the XML information server to the client that are really needed for query evaluation and that are not yet stored in the client's cache.

Finally, we have provided a performance evaluation that shows that a significant reduction in data exchange can be achieved by CSC.

An interesting extension of our research is to consider modification of compressed data on the client side or the server side. Modification of compressed XML data without complete decompression has been solved in (Böttcher and Steinmetz 2007b), and it seems to be promising to apply this to our compressed XML cache. Furthermore, cache updates and consistency between an XML server and an XML cache have been solved for uncompressed XML (Böttcher 2006).

Therefore, we consider it a promising challenge to combine all three aspects caching, compression and consistent updates in future work.

As XPath is used in other important XML standards like XSLT and XQuery, we consider it a challenging research topic to enhance the results presented here in such a way that they are applicable to XQuery or XSLT as well.

6 References

- Abiteboul, S., Segourin, L. and Vianu, V. (2001): Representing and querying XML with incomplete information. *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Santa Barbara, California, USA, ACM Press.
- Balmin, A., Özcan, F., Beyer, K.S., Cochrane, R. and Pirahesh, H. (2004): A Framework for Using Materialized XPath Views in XML Query Processing. (e)*Proceedings of the Thirtieth International Conference on Very Large Data Bases*, Toronto, Canada, 60-71, Morgan Kaufmann.
- Benedikt, M., Fan, W. and Geerts, F. (2005): XPath satisfiability in the presence of DTDs. *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, New York, NY, USA, 25-36, ACM Press.
- Böttcher, S., Hartel, R. and Heinzemann, C. (2008): BSBC: Towards a succinct data format for XML streams. *Proceedings of the Fourth International Conference on Web Information Systems and Technologies*, Funchal, Portugal, 13-21, INSTICC Press.
- Böttcher, S. and Steinmetz, R. (2007a): Evaluating XPath Queries on XML Data Streams. *Data Management. Data, Data Everywhere, 24th British National Conference on Databases*, Glasgow, UK, 101-113, Springer.
- Böttcher, S. and Steinmetz R. (2007b): Data Management for Mobile Ajax Web 2.0 Applications. *Database and Expert Systems Applications, 18th International Conference, DEXA 2007*, Regensburg, Germany, 424-433, Springer.
- Böttcher, S., Steinmetz, R. and Klein, N. (2007): XML index compression by DTD subtraction. *Proceedings of the Ninth International Conference on Enterprise Information Systems*, Funchal, Madeira, Portugal, 86-94.
- Böttcher, S. (2006): Cache Consistency in Mobile XML Databases. *Advances in Web-Age Information Management*, Hong Kong, China, 300-312, Springer.
- Böttcher, S. and Türling, A. (2004): Caching XML Data on Mobile Web Clients. *Proceedings of the International Conference on Internet Computing, IC '04*, Las Vegas, Nevada, USA, 150-156, CSREA Press.
- Buneman, P., Grohe, M. and Koch, C. (2003): Path Queries on Compressed XML. *Proceedings of 29th International Conference on Very Large Data Bases*, Berlin, Germany, 141-152, Morgan Kaufmann.
- Busatto, G., Lohrey, M. and Maneth, S. (2005): Efficient Memory Representation of XML Documents, *Database Programming Languages, 10th International Symposium*, Trondheim, Norway, 199-216, Springer.
- Cheney, J. (2001): Compressing XML with multiplexed hierarchical models. *Proceedings of the 2001 IEEE Data Compression Conference (DCC 2001)*, Snowbird, Utah, USA, 163-172, IEEE Computer Society.
- Cheng, J. and Ng, W. (2004): XQzip: Querying Compressed XML Using Structural Indexing. *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology*, Heraklion, Crete, Greece, 219-236, Springer.
- Hidders, J. (2003): Satisfiability of XPath expressions. *Database Programming Languages, 9th International Workshop, DBPL 2003*, Potsdam, Germany, 21-36, Springer.
- Koch, C., Scherzinger, S. and Schmidt M. (2008): XML Prefiltering as a String Matching Problem. *Proceedings of the 24th International Conference on Data Engineering*, Cancun, Mexico, 626-635, IEEE.
- Liefke, H. and Suciu, D. (2000): XMill: An Efficient Compressor for XML Data, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, USA, 153-164, ACM.
- Ng, W., Lam, W.Y., Wood, P.T. and Levene M. (2006): XCQ: A querifiable XML compression system. *Knowledge and Information Systems*, **10**(4):421-452.
- Mandhani, B. and Suciu, D. (2005). Query caching and view selection for XML databases. *Proceedings of the 31st international conference on Very large data bases*, Trondheim, Norway, 469-480, ACM.
- Marian, A. and Siméon, J. (2003): Projecting XML Documents. *Proceedings of 29th International Conference on Very Large Data Bases*, Berlin, Germany, 213-224, Morgan Kaufmann.
- Olteanu, D., Meuss, H., Furche, T. and Bry, F. (2002): XPath: Looking Forward. *XML-Based Data Management and Multimedia Engineering – EDBT*

- 2002 Workshops, Prague, Czech Republic, 109-127, Springer.
- O'Neil, P.E., O'Neil, E.J., Pal, S., Cseri, I., Schaller, G. and Westbury, N.(2004): ORDPATHs: Insert-Friendly XML Node Labels. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France, 903-908, ACM.
- Schmidt, A., Waas, F., Kersten, M., Carey, M., Manolescu, I. and Busse, R. (2002): XMark: A benchmark for XML data management. *Proceedings of 28th International Conference on Very Large Data Bases*, Hong Kong, China, 974-985, Morgan Kaufmann.
- Xu, W. and Ozsoyoglu, Z.M. (2005): Rewriting XPath queries using materialized views. *Proceedings of the 31st international conference on Very large data bases*, Trondheim, Norway, 121-132, ACM.
- Yang, L.H., Lee, M.-L. and Hsu, W. (2003): Efficient mining of XML query patterns for caching. *Proceedings of 29th International Conference on Very Large Data Bases*, Berlin, Germany, 69-80, Morgan Kaufmann.
- Zhang, N., Kacholia, V. and Özsu, M.T. (2004): A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. *Proceedings of the 20th International Conference on Data Engineering*, Boston, MA, USA, 54-65, IEEE Computer Society.