Combining Efficient XML Compression with Query Processing

Przemysław Skibiński[1] and Jakub Swacha[2]

[1] University of Wrocław, Institute of Computer Science,
Joliot-Curie 15, 50-383 Wrocław, Poland
inikep@ii.uni.wroc.pl
[2] The Szczecin University, Institute of Information Technology in Management,
Mickiewicza 64, 71-101 Szczecin, Poland
jakubs@uoo.univ.szczecin.pl

Abstract. This paper describes a new XML compression scheme that offers both high compression ratios and short query response time. Its core is a fully reversible transform featuring substitution of every word in an XML document using a semi-dynamic dictionary, effective encoding of dictionary indices, as well as numbers, dates and times found in the document, and grouping data within the same structural context in individual containers. The results of conducted tests show that the proposed scheme attains compression ratios rivaling the best available algorithms, and fast compression, decompression, and query processing.

Keywords: XML compression, XML searching, XML transform, semi-structural data compression, semi-structural data searching.

1 Introduction

Although Extensible Markup Language (XML) did not make obsolete all the good old data formats, as one could naively expect, it has become a popular standard, with many useful applications in information systems.

XML has many advantages, but for many applications they are all overshadowed by just one disadvantage, which is XML verbosity. But verbosity can be coped with by applying data compression. Its results are much better if a compression algorithm is specialized for dealing with XML documents.

XML compression algorithms can be divided into two groups: those which do not allow queries to be made on the compressed content, and those which do. The first group focuses on attaining as high compression ratio as possible, employing the state-of-the-art general-purpose compression algorithms: Burrows-Wheeler Transform – BWT [2], and Prediction by Partial Match – PPM [18]. The problem with the most effective algorithms is that they require the XML document to be fully decompressed prior to processing a query on it.

The second group sacrifices compression ratio for the sake of allowing search without a need for full document decompression. This can be accomplished by compressed pattern matching or by partial decompression. The former consists in compressing the pattern and searching for it directly in the compressed data. As both the compressed pattern and data are shorter than their non-compressed equivalents, the search time is significantly reduced. However, the compressed pattern matching works best with a context-free compression scheme, such as Huffman coding [9], which hurts compression ratio significantly.

Partial decompression means some parts of the document have to be decompressed, but not the whole of it. Context-aware algorithm can be used, as long as it makes use only of the limited-range data correlation. LZ77-family algorithms [21] are ideal candidates here. Although they do not attain as high compression ratios as BWT or PPM derivatives, they are much more effective than context-free compression schemes. And although they do not allow queries to be processed as fast as using compressed matches, the processing time is much shorter than in case of schemes requiring full decompression.

In this paper we describe a new algorithm designed with compression effectiveness as primary concern, and search speed as secondary one. Thus it is particularly suited for huge XML datasets queried with moderate frequency.

We begin with discussion of existing XML compression algorithms' implementations. Then we describe the proposed algorithm, its main ideas and its most significant details. Finally, we present results of tests of an experimental implementation of our algorithm, and draw some conclusions.

2 XML Compression Schemes

2.1 Non-query-supporting Schemes

The first XML-specialized compressor to noticeably surpass in efficiency general-purpose compression schemes was XMill [11]. Its success was due to three features. The first of them is splitting XML document content into three distinct parts containing respectively: element and attribute symbol names, plain text and the document tree structure. Every part has different statistical properties, therefore it helps compression to process them with separate models.

The second feature of XMill is to group contents of same XML elements into so-called containers. Thus similar data are stored together, helping compression algorithms with limited history buffer, such as LZ77 derivatives.

The third one is to encode each container using a dedicated method, exploiting the type of data stored within it (such as numbers or dates). What makes this feature not so useful is that XMill requires the user themself to choose methods to encode specific containers. Such human-aided compression can hardly be regarded as practical.

XMill originally used LZ77-derived gzip to compress the transform output. Although newer version added support for BWT-based bzip2 and PPM implementations, yet in these modes XMill succumbs to other programs employing such algorithms.

The first published XML compression scheme to use PPM was XMLPPM [3]. XMLPPM replaces element and attribute names with their dictionary indices, removes closing tags as they can be reconstructed in a well-formed XML document only provided their positions are marked. The most important XMLPPM feature is 'multiplexed hierarchical modeling' which consists in encoding data with four distinct PPM models: one for element and attribute names, one for element structure, one for attribute values, and one for element contents. In order to exploit some correlation between data going to different models, the previous symbol, regardless the model it belongs to, is used as a context for the next symbol.

XMLPPM was extended into SCMPPM [1], in which a separate PPM model is maintained for every XML element. This helps only in case of large XML documents, as every PPM model requires a due number of processed symbols to become effective. The main flaw of SCMPPM is its very high memory usage (every new model is initialized with 1 MB of allocated memory).

In 2004 Toman presented Exalt [20], one of the first algorithms compressing XML by inferring a context-free grammar describing its structure. His work led to AXECHOP by Leighton et al. [10]. However, even the latter fails to overcome XMLPPM both in terms of compression ratio and time.

In 2005 Hariharan and Shankar developed XAUST [8], employing finite-state automata (FSA) to encode XML document structure. Element contents are put into containers and encoded incrementally with arithmetic coding based on a single statistical model of order 4 (i.e., treating at most 4 preceding symbols as the context for the next one). The published results show XAUST to beat XMLPPM on some test files, yet XAUST has this great drawback that it requires the compressed XML document to be valid and its document type definition (DTD) to be available, as the FSA are constructed based on it.

2.2 Query-Supporting Schemes

As it has been stated in the introduction, adding support for queries decreases compression effectiveness. Therefore, most of the algorithms described below attain compression ratios worse than the original XMill.

XGrind [19] was probably the first XML compressor designed with fast query processing on mind. XGrind forms a dictionary of element and attribute names based on DTD. It uses a first pass over a document to gather statistics so that Huffman trees could be constructed. During the second pass, the names are replaced with respective codewords, and the remaining data are encoded using several Huffman trees, distinct for attribute values and PCDATA elements. An important property of XGrind, mainly thanks to the use of Huffman encoding, is that its transformation is homomorphic, which means that the same operations (such as parsing, searching or validating) can be performed on the compressed document as on its non-compressed form.

XPress [14] extends XGrind with binary encoding of decimal numbers and improves speed of path-based queries using a technique called Reverse Arithmetic Encoding.

XQzip [5] was the first query-supporting XML compressor to beat XMill, although slightly. It separates structure and data of XML documents. The structure is stored in a form of Structural Indexing Tree (SIT) in order to speed up queries, whereas data are grouped in containers, which are partitioned into small blocks (for the sake of partial decompression) and those are finally compressed with gzip. Recently decompressed blocks are kept in cache so that multiple queries are sped up. According to its authors, XQzip processes queries 12 times faster than XGrind.

XQueC [15] was clearly focused on search speed rather than compression efficiency. Like other schemes, it splits XML documents into structure and data parts, groups element contents in containers (based on each element's path from the document root), and forms a dictionary of element and attribute names. Additionally, for the sake of faster query processing, XQueC stores a tree describing document structure and an index – a structural summary representing all possible paths in the document.

XSeq [12] uses Sequitur, a grammar-based compression algorithm to compress both document structure and data, and allows compressed pattern matching. According to its authors, XSeq processes queries even faster than XQzip, but it does not attain compression ratios of XMill.

XBzip [7] employs XBW transform (based on BWT), which transposes XML document structure into its linear equivalent using path-sorting and grouping. Data are sorted accordingly with the structure. The two resulting tables (structure and data) are then compressed using PPM algorithm. XBzip can work in two modes. In the default, non-query-supporting mode, it can attain compression ratios even higher than XMLPPM. In the second, query-supporting mode (required by the XBzipIndex utility) it splits data into containers, and creates an FM-index (a compressed representation of a string that supports efficient substring searches) for each of them. The query processing times are very short, but storing the indices inflates the compressed document size by as much as 25-100%.

3 The QXT Transform

3.1 Overview

Our recent work on highly effective but non-query-supporting XML compression scheme dubbed XWRT [17] led us to awareness of the importance of fast query processing in XML documents. In practice, it is often desirable both to store data compactly, and retain swift query response time.

QXT stands for Query-supporting XML Transform. QXT has been designed combining the best solutions of XWRT with query-friendly concepts in order to make it possible to process queries with partial decompression, while avoiding to hurt compression effectiveness significantly.

For QXT, the input XML document is considered to be an ordered sequence of *n* tokens:

$$Input=(t_1 \cdot t_2 \cdot \ldots \cdot t_n).$$

QXT parses the input classifying every encountered token to one of generic token classes:

$$TokenClass(t) \in \{\ Word, EndTag, Number, Special, Blank, Char\ \}.$$

The *Word* class contains sequences of characters meeting the requirements for inclusion in the dictionary, *EndTag* contains all the closing tags, *Number* – sequences of digits, *Special* – sequences of digits and other characters adhering to predefined patterns, *Blank* – single spaces between *Word* tokens, and the *Char* class contains all the remaining input symbols.

The *Word* class has two token subclasses: *StartTag* contains all the element opening tags, whereas *PlainWord* all the remaining *Word* tokens.

The *StartTag* and *EndTag* tokens define the XML structure. The *StartTag* tokens differ from *PlainWord* tokens in that they redirect the transform output to a container identified by the opened element's path from the document root. *EndTag* tokens are replaced with a one-byte flag and bring the output back to the parent element's container.

3.2 Handling the Words

A sequence of characters can only be identified as a *Word* token if it is one of the following:

- *StartTag* token – a sequence of characters starting with '<', containing letters, digits, underscores, colons, dashes, or dots. If a *StartTag* token is preceded by a run of spaces, they are combined and treated as a single token (useful for documents with regular indentation);
- a sequence of lowercase and uppercase letters ('a'–'z', 'A'–'Z') and characters with ASCII codes from range 128–255; this includes all words from natural languages using 8-bit letter encoding;
- URL prefix – a sequence of the form 'http://domain/', where domain is any combination of letters, digits, dots, and dashes;
- e-mail – a sequence of the form 'login@domain', where 'login' and 'domain' are combinations of letters, digits, dots, and dashes;
- XML entity – a sequence of the form '&data;', where data is any combination of letters (so, e.g., character references are not included);
- attribute value delimiter – sequences '="' and '">';
- run of spaces – a sequence of spaces not followed by a *StartTag* token (again, useful for documents with regular indentation).

The list of *Word* tokens sorted by descending frequency composes the dictionary. QXT uses a semi-dynamic dictionary, that is it constructs a separate dictionary for every processed document, but, once constructed, the dictionary is not changed during XML transformation. It would be problematic to use a universal static dictionary with predefined list of words as it is hard to find a word set relevant across a wide range of real-world XML documents.

Every *Word* token is replaced with its dictionary index. The dictionary indices are encoded using symbols which are not existent in the input XML document. There are two modes of encoding, chosen depending on the attached back-end compression algorithm. In both cases, a byte-oriented prefix code is used; although it produces slightly longer output than, e.g., bit-oriented Huffman coding, the resulting data can be easily compressed further, which is not the case with the latter.

In the Deflate-friendly mode, the set of available symbols is divided into three disjoint subsets: *OneByte*, *TwoByte*, *ThreeByte*. The *OneByte* symbols are used to encode the most frequent *Word* tokens; one symbol can represent one token, so only

|*OneByte*| tokens can be encoded this way. The *TwoByte* symbols are used as a prefix for another byte, allowing to encode |*TwoByte*|·256 tokens in this way. Finally, the *ThreeByte* symbols are used as a prefix for another two bytes, allowing to encode |*ThreeByte*|·65536 tokens in this way.

In the LZMA-friendly mode, the set of available symbols is divided only into two disjoint subsets: *Prefix*, *Suffix*. The *Prefix* symbols signalize the beginning of a codeword. The codeword can be but a *Prefix* symbol, or a *Prefix* symbol followed by one or two *Suffix* symbols. This way there are |*Prefix*| one-byte codewords available for the most frequent *Word* tokens, |*Prefix*|·|*Suffix*| two-byte codewords for typical *Word* tokens, and |*Prefix*|·|*Suffix*|^2 three-byte codewords for rare *Word* tokens.

As the single *Blank* tokens can appear only between two *Word* tokens, they are simply removed, as they can be reconstructed on decompression provided the exceptional positions where they should not be inserted are marked.

The *Char* tokens are left intact.

3.3 Handling the Numbers and Special Data

Every *Number* token (decimal integer number) n is replaced with a single byte whose value is $\lceil \log_{256}(n+1) \rceil+48$. The actual value of n is encoded as a base-256 number. A special case is made for sequences of zeroes preceding another *Number* token – these are left intact.

Special token represent specific types of data made up of combination of digits and other characters. Currently, QXT recognizes following *Special* tokens:

- dates between 1977-01-01 and 2153-02-26 in YYYY-MM-DD (e.g. “2007-03-31”, Y for year, M for month, D for day) and DD-MMM-YYYY (e.g. “31-MAR-2007”) formats;
- times in 24-hour (e.g., “22:15”) and 12-hour (e.g., “10:15pm”) formats;
- value ranges (e.g., “115-132”);
- decimal fractional numbers with one (e.g., “1.2”) or two (e.g., “1.22”) digits after decimal point.

Dates are replaced with a flag and encoded as a two bytes long integer whose value is the difference in days from 1977-01-01. To simplify the calculations we assume each month to have 31 days. If the difference with the previous date is smaller than 256, another flag is used and the date is encoded as a single byte whose value is the difference in days from the previous date.

Times are replaced with a sequence of three bytes representing respectively: the time flag, hour, and minutes.

Value ranges in the format “x–y” where $x < 65536$ and $0 < y - x < 256$ are encoded in four bytes: one for the range flag, two for the value of x, and one for the difference $y - x$.

Decimal fractional numbers with one digit after decimal point and value from 0.0 to 24.9 are replaced by two bytes: a flag and their value stored as fixed point integer. In case of those with two digits after decimal point, only their suffix, starting from the decimal point, is considered to be *Special* token, and replaced with two bytes: a flag and the number’s fractional part stored as an integer.

3.4 Implementation

The architecture of QXT implementation is presented on Fig. 1.

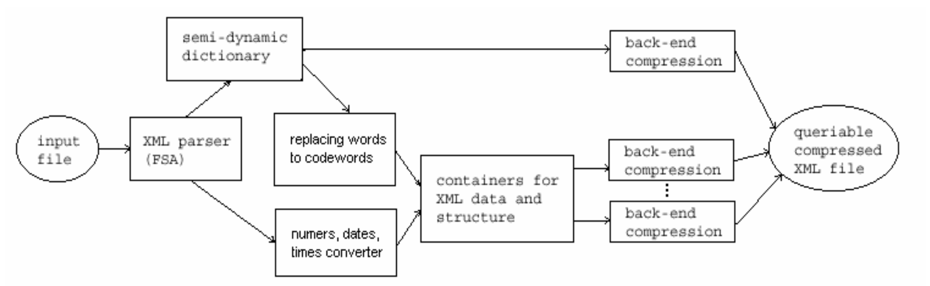
semi-dynamic dictionary
back-end compression
input file
XML parser (FSA)
replacing words to codewords
back-end compression
queriable compressed XML file
containers for XML data and structure
back-end compression
numers, dates, times converter

Fig. 1. QXT processing scheme

The QXT implementation contains a fast and simple XML parser written exclusively for this application. The parser does not build any trees, but it treats input XML document as one-dimensional data. It has small memory requirements, as it only uses a stack to trace opening and closing tags.

QXT works in two passes over input data. In the first pass, a dictionary is formed and the frequency of each of its items is computed. The complete dictionary is stored within the compressed file, so this pass is unnecessary during decompression, making the reverse operation faster.

In the second pass, the actual transform takes place, data are parsed into tokens, respectively encoded, and placed into separate containers, depending on their path from the document root.

The containers are memory-buffered until they exceed a threshold of 8 MB (same as in XMill) when they are compressed with general-purpose compression algorithm and written to disk. The containers are compressed in 32 KB blocks, thus allowing partial decompression of blocks of such length.

QXT could be combined with any general-purpose compression algorithm, but the requirement of fast decompression suggests using a LZ77-derivative. We chose two algorithms of this kind: Deflate [6] (well known from zip, gzip, and plenty of other applications) and LZMA (known from the 7-zip utility [22]), whose optimal match parsing significantly improves compression ratio at the cost of much slower compression (decompression speed is not much affected).

Query execution starts with reading the dictionary from the compressed file; the dictionary can be cached in memory in case of multiple queries. Next, the query processor resolves which containers might contain data matching the query. The required containers are decompressed with the general-purpose algorithm, and the transformed data are then searched using the transformed pattern. Only the matching elements are decoded to the original XML form; of course, counting queries do not require the reverse transform at all.

4 Experimental Results

4.1 Test Preparation

The primary objective of the tests was to measure the performance of an experimental implementation of the QXT algorithm written in C++ by the first author and compiled with Microsoft Visual C++ 6.0. This implementation allows to use Deflate or LZMA as the back-end compression algorithm.

For comparison purposes, we included in the tests publicly available XML compressors: XMill (version 0.7, which was found to be the fastest; switches: -w -f), XMLPPM (0.98.2), XBzip (1.0), and SCMPPM (0.93.3); the old XGrind was omitted as, despite multiple efforts, we were not able to port it to the test system. We have extended the list with general-purpose compression tools: gzip (1.2.4; uses Deflate) and LZMA (4.42; -a0), employing the same algorithms as the final stage of QXT, to demonstrate the improvement from applying the XML transform.

So far there is no publicly available and widely respected XML test file corpus, therefore, we have based our test suite on those files from the corpus proposed in [15] that were publicly available, improving it with several files from the University of Washington *XML Data Repository* [13]. The resulting corpus represents a wide range of real-world XML applications; it consists of the following varied XML documents:

- *DBLP*, bibliographic information on major computer science journals and proceedings,
- *Lineitem*, business order line items from the 10 MB version of the TPC-H benchmark,
- *Mondial*, basic statistical data on countries of the world,
- *NASA*, astronomical data,
- *Shakespeare*, a corpus of marked-up Shakespeare plays,
- *SwissProt*, a curated protein sequence database,
- *UWM*, university courses.

Detailed information for each of the documents is presented in Table 1 (see next page); it includes: file size (in bytes), number of elements, number of attributes, number of distinct element types, and the maximum structure depth.

The tests were conducted on an Intel Core 2 Duo E6600 2.40 GHz system with 1024 MB memory and two Seagate 250 GB SATA drives in RAID mode 1 under Windows XP 64-bit edition.

Table 1. Basic properties of the XML documents used in tests

Name	File size	Elements	Attributes	Max. depth
DBLP	133 862 735	3 332 130	404 276	6
Lineitem	32 295 475	1 022 976	1	3
Mondial	1 784 825	22 423	47 423	5
NASA	25 050 288	476 646	56 317	8
Shakespeare	7 894 983	179 690	0	7
SwissProt	114 820 211	2 977 031	2 189 859	5
UWM	2 337 522	66 729	6	5

4.2 Compression Ratio and Time

Table 2 lists the bitrates (in output bits per input character, hence the smaller the better) attained by the tested programs on each of the test files. Some table cells are empty, as XMill and SCMPPM declined to compress some of the test files, whereas XBzip failed to finish the compression of two files due to insufficient memory.

Apart from the general-purpose compression tools, the experimental QXT implementation was the only program to decode all the compressed files accurately. In case of the other programs, some output files were shortened, some had misplaced white characters (space-preserving modes were turned on where possible). In some applications, even the latter can be a grave flaw, as this makes it impossible to verify the original file integrity with a cyclic redundancy check or hash functions.

Table 2. Compression results (in bpc)

File	Gzip	LZMA	XMill	XML PPM	SCM PPM	XBzip	XBzip Index	QXT Deflate	QXT LZMA
DBLP	1.463	1.049	1.250[a]	0.857[a]	**0.741**	–[d]	–[d]	0.925	0.753
Lineitem	0.721	0.461	0.380	0.273[a]	**0.244**	0.248[a]	0.332	0.285	0.245
Mondial	0.767	0.586	–[c]	0.467[b]	–[c]	**0.404**[a]	0.681	0.608	0.407
NASA	1.208	0.818	1.011	0.729[a]	–[c]	0.698[b]	1.085	0.769	**0.607**
Shakespeare	2.182	1.786	2.044	1.367[a]	1.354[a]	**1.350**[b]	1.688	1.505	1.354
SwissProt	0.985	0.540	0.619	0.465[a]	0.426	–[d]	–[d]	0.500	**0.384**
UWM	0.553	0.389	0.382	**0.259**[a]	0.274	0.282	0.446	0.323	0.281
Average	1.126	0.804	–	0.631	–	–	–	0.702	**0.576**

Remarks: (a) Decoded file was not accurate, (b) Decoded file was shorter than the original, (c) Input file was not accepted, (d) Compression failed due to insufficient memory.

The obtained compression results are very favorable for QXT. The QXT+LZMA attained the best ratio in case of two files and was only slightly worse than the best scheme in case of four other files. Notice that this was achieved even though QXT supports queries, whereas the other programs do not (with the sole exception of XBzipIndex), and it uses LZ77-derived compression algorithm, whereas the best of the remaining programs employ more sophisticated PPM-based algorithms.

Compared to the general-purpose compression tools, the proposed transform improves XML compression on average by 28% in case of LZMA and 37% in case of Deflate.

Fig. 2 shows the ratio of the compressed file size to the original size for the four biggest files in the suite, attained by the tested XML-specialized compressors. Smaller columns represent better results.

Table 3 contains the compression and decompression times measured on *Lineitem*, the longest file compressed by all the tested programs.

The results show that QXT is slower than XMill, but when compared to the most effective compressors, QXT+Deflate was found to be almost about three times faster than XBzip, and seven times faster than SCMPPM.

In case of QXT+LZMA, the XML transform is faster than the back-end compression algorithm, and reduces the file size so much, that a large improvement in compression speed can be observed.

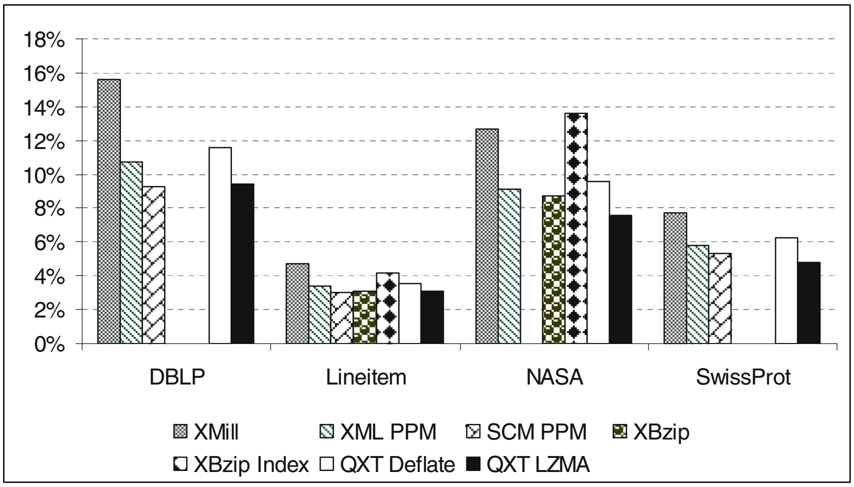

Fig. 2. Compression results for selected files

Table 3. Compression and full decompression times for the *Lineitem* file

Time of	Gzip	LZMA	XMill	XML PPM	SCM PPM	XBzip	XBzip Index	QXT Deflate	QXT LZMA
Compre-ssion	0.98	23.50	1.31	2.66	14.24	6.33	11.59	3.06	5.95
Decom-pression	0.20	0.83	0.22	3.19	13.16	4.51	6.26	1.12	1.34

Remarks: The times listed were measured on the test platform, and are total user times including program initialization and disk operations.

4.3 Query Processing Time

For query processing evaluation, we used the *Lineitem* and *Shakespeare* files (we could not obtain results for XBzipIndex on *DBLP*). Queries *S1*, *S2*, and *L1* are from [5], but as XBzipIndex did not support another query, we contrived the remaining queries on our own. Table 4 lists all the queries used in our tests.

Table 4. Test queries

Id	*Shakespeare* queries
S1	/PLAY/ACT/SCENE/SPEECH/SPEAKER
S2	/PLAY/ACT/SCENE/SPEECH[SPEAKER = "PHILO"]
S3	/PLAY/ACT/SCENE/SPEECH[SPEAKER = "AMIGA"]
Id	*Lineitem* queries
L1	/table/T/L_TAX
L2	/table/T/L_COMMENT
L3	/table/T/[L_COMMENT = "slowly"]

Table 5 contains measured times of processing the test queries. Its first column identifies the query. The next two columns contain query processing time on a non-compressed XML without an external index (column 2), and using one (column 3). The sgrep structural search utility (version 1.92a) was used to obtain these values. Column 4 contains XBzipIndex's query time, and the remaining columns present detailed results for QXT: the part of the file (in percents) that had to be decompressed in the course of query processing (column 5), the time it took to accomplish decompression (column 6 for Deflate, and 7 for LZMA), the time taken by the actual search on the decompressed data (column 8), and the sum of the two (columns 9 and 10), constituting the total query processing time.

Table 5. Query processing times (in seconds)

Id	sgrep		XBzip Index	QXT					
	Raw XML	Inde-xed XML		Decom-pressed part	Decompression		Search time	Total query time	
					Deflate	LZMA		Deflate	LZMA
S1	0.313	0.047	0.047	15.7%	0.051	0.025	0.023	0.074	0.048
S2	0.282	0.032	0.061	15.7%	0.051	0.025	0.023	0.074	0.048
S3	0.266	0.031	0.063	10.9%	0.036	0.017	0.000	0.036	0.017
L1	1.031	0.063	0.047	5.5%	0.062	0.083	0.036	0.098	0.119
L2	1.063	0.078	0.047	8.4%	0.095	0.127	0.055	0.150	0.182
L3	1.047	0.047	0.063	8.4%	0.095	0.127	0.055	0.150	0.182
Avg	0.667	0.050	0.055	10.8%	0.065	0.067	0.032	0.097	0,099

Remarks: The times listed were measured on the test platform, and are total user times including program initialization and disk operations.

Average query processing time of QXT was about 0.1 second. This seems to be an acceptable delay for an average user. This is over six times faster than the raw XML search time. The transformed data are searched much faster than non-compressed XML because they are shorter. The decompression does not waste all of this gain because an average query requires only 10% of the file to be decompressed. Moreover, the word dictionary makes it possible to immediately return negative results for queries looking for words non-occurring in the document (see the instant search results for query *S3*).

The primary purpose of QXT is effective compression, so it does not maintain any indices to the document content. Therefore, we did not expect QXT to beat index-based search times. Indeed, both sgrep's indexed search and XBzipIndex are over 50% faster. However, a look at the compression ratio of XBzipIndex, signifying a 25% bigger storage requirements than QXT+LZMA, helps to realize that these two schemes pursue two different goals.

To demonstrate the scale of improvement over a non-query-supporting schemes that perform full decompression followed by search on the decompressed document, QXT's average query time on *Shakespeare* (0.037 s) is over ten times shorter than XMill's (0.381 s) and over hundred times shorter compared to the time required by SCMPPM (3.834 s).

5 Conclusions

Due to the huge size of XML documents used nowadays, their verbosity is considered a troublesome feature. Until now, there have been but two options: effective compression for the price of a very long data access time, or quickly accessible data for the price of mediocre compression.

The proposed QXT scheme ranks among the best available algorithms in terms of compression efficiency, far surpassing its rivals in decompression time.

The most important advantage of QXT is the feasibility of processing queries on the document without a need to have it fully decompressed. Thanks to the nature of the transform, the measured query processing times on QXT-transformed documents were several times shorter than on their original format. We did not include any indices in QXT as they require significant storage space, so using them would greatly diminish the compression gain which had the top priority in the design of QXT. Still, we may reconsider this idea in a future work.

QXT has many nice practical properties. The transform is completely reversible, the decoded document is an accurate copy of the input document. The transform requires no metadata (such as XML Schema or DTD) nor human assistance. Whereas SCMPPM and XBzip may require even hundreds of megabytes of memory, the default mode of QXT uses only 16 MB, irrespectively of the input file size (using LZMA requires additionally a fixed buffer of 84 MB for compression and 10 MB for decompression). The compression is done on small blocks, so in case of data damage, the data loss is usually limited to just one such block. Moreover, QXT is implemented as a stand-alone program, requiring no external compression utility, XML parser, nor query processor, thus avoiding any compatibility issues.

Acknowledgements

Szymon Grabowski is the co-author of XWRT, the non-query-supporting XML compression scheme on which QXT has been based.

References

1. Adiego, J., de la Fuente, P., Navarro, G.: Merging Prediction by Partial Matching with Structural Contexts Model. In: Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, p. 522 (2004)
2. Burrows, M., Wheeler, D.J.: A block-sorting data compression algorithm. SRC Research Report 124. Digital Equipment Corporation, Palo Alto, CA, USA (1994)
3. Cheney, J.: Compressing XML with multiplexed hierarchical PPM models. In: Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, pp. 163–172 (2001)
4. Cheney, J.: Tradeoffs in XML Database Compression. In: Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, pp. 392–401 (2006)
5. Cheng, J., Ng, W.: XQzip: querying compressed XML using structural indexing. In: Proceedings of the Ninth International Conference on Extending Database Technology, Heraklion, Greece, pp. 219–236 (2004)

6. Deutsch, P.: DEFLATE Compressed Data Format Specification version 1.3. RFC1951(1996), http://www.ietf.org/rfc/rfc1951.txt
7. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and Searching XML Data Via Two Zips. In: Proceedings of the International World Wide Web Conference (WWW), Edinburgh, Scotland, pp. 751–760 (2006)
8. Hariharan, S., Shankar, P.: Compressing XML documents with finite state automata. In: Farré, J., Litovsky, I., Schmitz, S. (eds.) CIAA 2005. LNCS, vol. 3845, pp. 285–296. Springer, Heidelberg (2006)
9. Huffman, D.A.: A Method for the Construction of Minimum-Redundancy Codes. Proc. IRE 40, 9, 1098–1101 (1952)
10. Leighton, G., Diamond, J., Muldner, T.: AXECHOP: A Grammar-based Compressor for XML. In: Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, pp. 467–467 (2005)
11. Liefke, H., Suciu, D.: XMill: an efficient compressor for XML data. In: Proceedings of the 19th ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, pp. 153–164 (2000)
12. Lin, Y., Zhang, Y., Li, Q., Yang, J.: Supporting efficient query processing on compressed XML files. In: Proceedings of the ACM Symposium on Applied Computing, Santa Fe, NM, USA, pp. 660–665 (2005)
13. Miklau, G.: XML Data Repository, University of Washington (2004), http:// www.cs.washington.edu/research/xmldatasets/www/repository.html
14. Min, J.-K., Park, M., Chung, C.: A Compressor for Effective Archiving, Retrieval, and Updating of XML Documents. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, pp. 122–133 (2003)
15. Ng, W., Lam, W.-Y., Cheng, J.: Comparative Analysis of XML Compression Technologies. World Wide Web 9(1), 5–33 (2006)
16. Skibiński, P., Grabowski, S., Deorowicz, S.: Revisiting dictionary-based compression. Software – Practice and Experience 35(15), 1455–1476 (2005)
17. Skibiński, P., Grabowski, S., Swacha, J.: Fast transform for effective XML compression. In: Proceedings of the IXth International Conference CADSM 2007, pp. 323–326. Publishing House of Lviv Politechnic National University, Lviv, Ukraine (2007)
18. Shkarin, D.: PPM: One Step to Practicality. In: Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, pp. 202–211 (2002)
19. Tolani, P., Haritsa, J.: XGRIND: a query-friendly XML compressor. In: Proceedings of the 2002 International Conference on Database Engineering, San Jose, CA, USA, pp. 225–234 (2002)
20. Toman, V.: Syntactical compression of XML data. In: Presented at the doctoral consortium of the 16th International Conference on Advanced Information Systems Engineering, Riga, Latvia (2004), http://caise04dc.idi.ntnu.no/CRC_CaiseDC/ toman.pdf
21. Ziv, J., Lempel, A.: A Universal Algorithm for Sequential Data Compression. IEEE Trans. Inform. Theory 23, 3, 337–343 (1977)
22. 7-zip compression utility, http://www.7-zip.org