

# YAQCX: A Word-based Query-aware Compressor for XML Data

Juliano Palmieri Lage<sup>1</sup>, Alberto H. F. Laender<sup>1</sup>, Edleno Silva de Moura<sup>2</sup>

<sup>1</sup>Department of Computer Science – Federal University of Minas Gerais  
31270-901 Belo Horizonte MG Brazil

<sup>2</sup>Department of Computer Science – Federal University of Amazonas  
69077-000 Manaus AM Brazil

{palmieri, laender}@dcc.ufmg.br, edleno@dcc.ufam.edu.br

**Abstract.** XML has become a *de facto* standard for data exchanging over the Internet. However, efficiently storing and querying XML data is still an open problem. In this paper we present YAQCX, Yet Another Query-aware Compressor for XML. YAQCX adopts word-based modeling combined with byte-coding to provide a very efficient approach to compressing/decompressing and querying XML data. It also implements a subset of XPath with a powerful pattern matching extension that allows regular expressions, range queries, and partial matching. Additionally, when processing queries, it accesses the actual compressed data as few as possible, for example to solve predicates on contents or to show results. Based on our experiments, we show that YAQCX compression ratios are comparable to XMill's and very close to those of other query-aware compressors, such as XQzip and XGrind. We also show that YAQCX compresses and decompresses faster than XMill, and outperforms XGrind regarding query processing.

## 1. Introduction

The eXtensible Markup Language (XML) [Bray et al. 2000] has become a *de facto* standard for data integration and exchange over the Internet, mainly due to its self-describing nature that eases data structuring and processing. However, XML presents as a major drawback its verbosity which makes storing and querying XML documents a major research issue today.

The storage problem can be easily solved by using general purpose compressors, such as `gzip`, or XML-oriented compressors, such as XMill [Liefke and Suciu 2000] and MHM [Cheney 2001]. These compressors, however, aim only at reducing storage space, but storage is just one side of the problem, since we need to query the stored documents too. To overcome this situation, several efforts have been made to deploy new techniques to make it possible to query directly over compressed XML documents [Arion et al. 2004, Cheng and Ng 2004, Lin et al. 2005, Min et al. 2003, Tolani and Haritsa 2002].

In this paper, we present YAQCX — Yet Another Query-aware Compressor for XML. YAQCX uses a new strategy to compress and search on compressed XML documents that has been derived from a method originally proposed to deal with natural language texts. This strategy allows YAQCX to search directly on compressed XML documents in a very efficient way. As shown by our experiments, YAQCX is able to process

queries 10 times faster than XGrind [Tolani and Haritsa 2002]. In addition to that, it is very fast to decompress, being 4.16 times faster than XMill [Liefke and Suciu 2000], and allows decompressing any specific fragment of a compressed XML document. The advantages of YAQCX when compared to previously proposed methods come from a new structure encoding scheme that represents each distinct path of an XML document by a unique code in the compressed text. This scheme turns decompression faster and yields significant performance improvements at query processing time.

Another important feature of YAQCX is its flexibility when processing queries. It implements a subset of XPath [Clark and DeRose 1999], which has been extended with a powerful pattern matching engine based on text compressing techniques proposed in [de Moura et al. 2000]. This combination makes YAQCX able to process word- and phrase-based searching and more flexible pattern matching facilities, such as character ranges, complements, wildcards, and arbitrary regular expressions, exactly or allowing errors, without decompressing the XML documents. Moreover, YAQCX is able to process phrase searching allowing insertion, deletion or replacement of words also without any decompression. To the best of our knowledge, these facilities are not allowed by any previously proposed XML query-aware compressor.

The rest of this paper is organized as follows. In Section 2 we discuss related work. Our compression approach is presented in Section 3 whilst querying processing over compressed data is discussed in Section 4. Our experimental results are presented in Section 5 where we compare YAQCX performance regarding compression ratio and query processing with similar approaches. Finally, in Section 6 we present our conclusions and directions for future work.

## **2. Related Work**

XMill [Liefke and Suciu 2000] was the first compressor designed specifically to deal with XML data, followed by MHM [Cheney 2001]. XMill divides XML documents into data containers that are compressed separately with a user assigned algorithm. Then, a general purpose compressor (`gzip` or `bzip2`) is used to shrink the remaining data. MHM multiplexes several text compression models based on XML syntactic structure and uses Prediction by Partial Matching for coding, achieving compression ratios 30% better than XMill, being, however, fairly slow.

Several approaches have been proposed to querying directly over compressed data and, as far as we know, the work presented in [de Moura et al. 2000] is still the one which provides the best tradeoff between compression ratio and performance. It proposes a word-based variation of the Huffman algorithm in which coding is byte-oriented instead of bit-oriented. Using a word-based model greatly improves the compression ratio while byte coding allows querying the compressed data with no decompression except that required to show the results. Moreover, this strategy provides more flexible pattern matching facilities, such as regular expressions, matches allowing errors, etc.

The first compressor to allow querying directly on compressed XML data was XGrind [Tolani and Haritsa 2002]. It allows exact- and prefix-matching queries, using dictionary encoding for tags and enumerated data values, and classical Huffman coding for non-enumerated data values. An XGrind compressed document remains homomorphic in relation to its uncompressed version, which allows querying in the same way that

in the uncompressed universe. XPRESS [Min et al. 2003], another homomorphic compressor, uses reverse arithmetic encoding to encode label paths as intervals, and allows range queries over the structure to be evaluated efficiently. XPRESS uses distinct encoding methods for each type of data value, which are inferred automatically.

More recently, new approaches have been proposed: XQzip [Cheng and Ng 2004], XQueC [Arion et al. 2004], and XSeq [Lin et al. 2005]. XQzip uses an index structure to remove redundant parts of an XML document, which reduces the query search space and improves query performance, avoiding full decompression by using a smart division of the XML document into small data blocks. It supports a wide range of XPath facilities, processing queries in polynomial time in the average case. XQueC implements XQuery processing over compressed XML data. Structure is stored separated from data which are also divided into containers and compressed with the most suitable algorithm, taking into account the query workload. The data fragmentation strategy used by XQueC grants in-memory query processing, however, the data structures used are too large and the compression ratios achieved are just slightly better than those of XPRESS. XSeq proposes a compression scheme based on the Sequitur algorithm for compressing text strings. Like XMill, XSeq first separates an XML document into data containers. Each container is then compressed using Sequitur, which produces a set of context-free grammar rules. The compressed result contains both the set of rules for each container and the necessary indices to help processing XPath queries. This scheme achieves compression ratios comparable to gzip and query performance comparable to XQzip.

In [Buneman et al. 2003], the authors address the problem of compressing the structure of XML documents using techniques borrowed from symbolic model checking. However, compressing only the document structure is not enough for solving the storage problem, since structure usually requires less storage space than content. For instance, in the XML datasets we used in our experiments, structure requires no more than 30% of the total amount of space spent to storing contents.

For a more comprehensive comparative analysis of existing XML compression techniques we refer the reader to [Ng et al. 2006].

### **3. Compressing XML Data**

YAQCX divides an XML document into containers aiming at lowering the model *entropy* [Bell et al. 1990] and, therefore, improving the compression ratio. The idea here is that redundancy among values inside the same element are likely to be greater than among values inside distinct elements. It uses distinct models for structure and contents to aid query processing and to improve compression ratios since we can choose the most suitable model for each case.

Following the work in [de Moura et al. 2000], YAQCX uses byte-oriented coding to achieve fast compression and decompression times, since bit shifting and masking are not necessary, and the spaceless word-based model to improve compression ratios and allow direct and efficient query processing over the compressed data. The spaceless word-based model uses words as symbols instead of characters, where a *word* is a sequence of non-space characters, and implicitly represents the most common separator (often a single space) alternating it with words and explicitly encoding other separators.

### 3.1. Structure Compression

One of the major contributions of YAQCX is the way it encodes the structure of the documents in order to improve querying performance. The idea is to keep a summary of the structure in memory that allows to process part of the queries without accessing the compressed data.

Our structure summary is a simple data structure in which each distinct path from root to element/attribute is represented by a unique ID assigned by a breadth first numbering over the document tree, keeping only information regarding the relationship among nodes. Thus, our structure summary requires a negligible storage space. A pointer to each parent element completes our structure summary. Combining the pointers and the numbering, we can easily keep track of the relation among ascendants, descendants and siblings. Moreover, we are able to efficiently process XPath axes in main memory without the need of accessing the compressed data to identify the relation among document nodes.

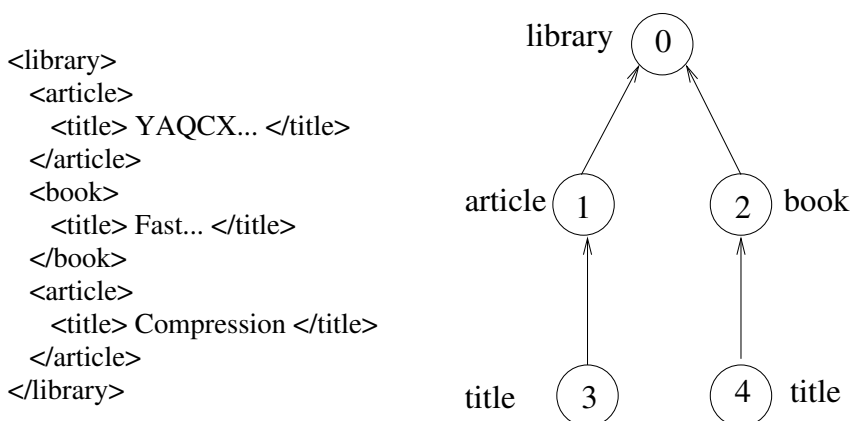


Figure 1. Structure summary.

YAQCX uses a dictionary to encode each path found in the summary. It assigns a unique identifier for each distinct node and uses a fixed size coding for representing each identifier in the compressed document. Due to byte-coding, each element or attribute identifier takes  $\lceil \log_{256}^t \rceil$  bytes, where  $t$  is the number of distinct nodes found in the summary.

Using this coding scheme, tags with the same name, but with different paths, are represented with different symbols in the compressed text. For instance, in Figure 1 the tag `title` is coded as 3 when it is a child of `article`, and is encoded as 4 when it is a child of `book`. This distinction can be seen as a query disambiguation technique and is useful for speeding up query processing because it avoids the need of keeping track of the context when processing path queries.

### 3.2. Contents Compression

Compression of data values is carried out by using the byte Huffman coding and the spaceless word-based model. In this scenario, each word or separator of the text is considered as a symbol and each codeword is a sequence of bytes. This combination was chosen considering that the YAQCX data structures include the vocabulary of the text, which is used by the compressed matching algorithm. Moreover, decompression is fast and we can

directly access any part of the compressed document without the need of decompressing the whole document, which are important issues when dealing with query processing over compressed data.

YAQCX uses a class of byte Huffman coding named *Tagged Huffman Code*, a coding scheme in which only the 7 less significant bits of each byte are used. The first byte of each codeword has the most significant bit set to 1, while the remaining ones have it set to 0. This aids directly searching the compressed text and plays an important role on the combination of structure and contents.

In addition, YAQCX groups all data under the same element name into homogeneous containers, assigning a Huffman tree for each element containing data (text nodes). Each Huffman tree has a vocabulary associated to it, which is accessed by hashing and contain all words of that container. Ideally the intersection among these vocabularies should be empty, but it is not what actually happens with real XML documents. Despite this, the intersection tends to be small.

### **3.3. Putting All Together**

Each dictionary encoded tag (start tag, end tag or attribute name) is preceded by escape sequences that are not present in any Huffman tree. There is one Huffman tree to store attribute values and spacing among the tags (indentation), and one Huffman tree for each nonempty element. We group all attribute values into the same tree, since they often are small, thus it is not worth dividing them into distinct trees. Furthermore, we group elements by name, mainly because element names usually are used in the same context and when they are used in distinct ones this arrangement plays a good tradeoff over entropy.

The escape sequences follow naturally from the tagged Huffman coding. We use two bytes with the most significant bits set to 1 as the sequence. Using 255 (0xFF in hexadecimal) — last node of the first level of the tree with the most significant bit set to 1 — as the first byte and another value greater than 127 (0x7F) as the second byte solves the problem. This sequence does not appear in a tagged Huffman coding.<sup>1</sup> YAQCX uses the second byte of the escape sequence to identify which kind of tag is being escaped, e.g., a start or end tag. This escaping scheme eases and greatly improves query processing, since YAQCX can preprocess the queries against the dictionary and, therefore, need not keep track of the context being searched.

Notice that a YAQCX compressed document is homomorphic with respect to the uncompressed document which allows the use of techniques available for uncompressed documents.

## **4. Querying over Compressed XML Data**

YAQCX supports most XPath [Clark and DeRose 1999] features and adds powerful pattern matching operators to it, combining them with the `contains` and `starts-with` functions. It is important to notice that the principles stated here can be used as building blocks for more expressive languages, such as XQuery.

---

<sup>1</sup>Easy to verify considering the three possible cases separately: (i) when the number of symbols are less than 128, (ii) equal to 128 and (iii) greater than 128.

## 4.1. Processing XPath

YAQCX query evaluation is divided into two distinct phases. In the first one, YAQCX uses the structure summary to process the location path identifying the type of node to be matched against the compressed data, and accesses the compressed document to process predicates when needed. During the second phase, YAQCX uses intermediate results evaluated in the previous phase to output the final results.

### 4.1.1. Location Paths

Our YAQCX prototype supports arbitrary nesting of predicates and 10 out of 13 XPath axes. We have not implemented the `following` and `preceding` axes and, since we do not model namespace nodes, we do not implement the `namespace` axis either. Our location step algorithm is the generic one described and analyzed in [Gottlob et al. 2002]. However, we need not traverse the entire document on each iteration, which means a huge gain in performance.

When processing a location step, the axis can be entirely evaluated using only the structure summary. We say that all nodes that fit under the same root-to-node path share the same *node type*. A node type has a type ID (tid) that is the combination of its tag identifier and the tag identifier of its parent (provided by the tag dictionary)<sup>2</sup>. Thus, for example, when processing the location path `/library/article` for the XML document in Figure 1, YAQCX uses the structure summary to discover that all nodes of type `article` fit this axis.

### 4.1.2. Processing Predicates

For processing predicates, YAQCX needs to access the compressed data (with no decompression at all) to select the actual nodes that evaluate for true under the node test. Using the node type information gathered during the axis processing, YAQCX generates the escape sequence of the element tag and searches it in the text using a simple text matching algorithm. Whenever a tag is found, the predicate condition is evaluated. If it is a contents predicate, techniques presented in Section 4.2 are used, if it is another location path the whole process is recursively repeated.

The result of each predicate is a set of nodes represented by a triple  $\langle \text{tid}, \text{start}, \text{end} \rangle$ , where ‘start’ and ‘end’ indicate the initial and final position of the node in the compressed document. This is the only data that come to main memory after the predicate processing.

### 4.1.3. Showing Results

When the processing phase ends, we have a set of triples or a set of node type identifiers if there are no predicates. In the first case, we use the ‘start’ information to locate the node and decompress it until its end tag, given by ‘end’.

---

<sup>2</sup>An easy induction proves that this pair is enough to uniquely identify a node type.

When we have only the node type identifier, YAQCX traverses the compressed document looking for its escape sequences. Whenever they are found, YAQCX starts decompressing the node until its end. Notice that YAQCX is able to start decompressing from any point of the compressed document, due to the modeling and coding we have chosen.

#### 4.2. Flexible Pattern Matching

Besides XPath processing, a major feature of YAQCX is its ability of processing flexible pattern matching over the contents of compressed documents. As stated before, YAQCX uses Tagged Huffman coding [de Moura et al. 2000] to compress contents, what allows the use of any classical text-searching algorithm to search directly over the compressed contents. YAQCX only needs to compress the words being searched, and then use the compressed pattern to search over the compressed document.

YAQCX is able to process *phrase patterns*, that is a sequence of words or *extended patterns*, patterns matched against single words including operations with characters in any position (range, arbitrary set, complement, wild card or case-insensitive patterns), regular expressions, approximate searching (insertion, deletion or replacement of characters), and combinations of them.

Processing these patterns over contents is divided into two phases. A preprocessing phase, during which the original pattern is converted into a set of simple compressed patterns. Patterns which do not match the vocabulary are discarded in this phase without requiring any access to the compressed document. The second one is the searching phase, during which YAQCX searches over the compressed document for the compressed patterns. When dealing with exact matches and simple patterns the preprocessor generates one single compressed pattern to be searched. When dealing with extended patterns and search allowing errors, all possibilities of matching over the vocabulary are generated and compressed, thus the original search task is converted into a multi-pattern search task. In these cases YAQCX uses a multi-pattern algorithm to search the contents.

### 5. Experimental Results

We evaluated YAQCX over a set of XML documents most of which have been used by other XML compressors, aiming at cross-comparing with them, since only XMill and XGrind are publicly available. The main characteristics of these XML datasets are presented in Table 1.

Dataset	Size (MB)	Distinct Paths	Distinct Tags	Data Containers
XMark	111	512	83	37
DBLP	153	133	41	27
SwissProt	109	131	99	13
Shakespeare	7.5	57	21	12
MeSH	228	111	74	33
Old Testament	3.3	27	21	13

Table 1. Datasets used in our experiments.

XMark [Schmidt et al. 2002] is a synthetic dataset that models an auction database; DBLP is an XML version of the world-wide known bibliography database; SwissProt is a DNA sequences dataset; Shakespeare is a set of tagged plays; MeSH, which stands for Medical Subject Headings, is an XML marked-up controlled vocabulary used for biomedical and health-related information; and Old Testament is a marked-up bible text.

YAQCX was implemented in C++ and all experiments were run on a stand-alone 1.5 GHz Pentium 4 with 256 MBytes of main memory, running SuSE linux.

### 5.1. Storing the Vocabulary

According to the Heaps' Law [Heaps 1978], a vocabulary grows sub-linearly in relation to the size of the text, that is, the size of the vocabulary is given by  $\mathcal{O}(u^\beta)$ , where  $u$  is the size of the text and  $0 < \beta < 1$  is a text dependent constant.

Table 2 shows the vocabulary characteristics of the datasets we used. As shown, the vocabulary usually is far smaller than the original document. The only exception was for DBLP due to a key element that uniquely identifies each publication entry. The  $\beta$  values indicate that the vocabularies grow at a lower rate when compared to the original documents. This also means that the ratio between the vocabularies and the original documents tends to decrease as the documents become larger. Thus, to consider that the vocabulary fits into memory is not a strong restriction.

Data Set	Corpus		Unique Vocabulary		Distinct Vocabularies		$\beta$
	Size (MB)	#Words	Size (MB)	#Words	Size (MB)	#Words	
XMark	111	16,993,881	1.33 (1.2%)	129,535 (0.7%)	1.86 (1.67%)	223,579 (1.3%)	0.421
DBLP	153	29,298,017	15.59 (10.1%)	1,099,001 (3.8%)	16.03 (10.5%)	1,106,042 (3.8%)	0.819
SwissProt	109	23,313,887	5.18 (4.7%)	548,717 (2.4%)	5.39 (4.9%)	553,758 (2.4%)	0.856
Shakespeare	7.5	1,472,854	0.21 (2.8%)	28,358 (1.9%)	0.25 (3.3 %)	32,949 (2.2%)	0.517
MeSH	228	20,555,087	2.4 (1.0%)	280,274 (1.3%)	3.09 (1.4%)	361,171 (1.7%)	0.824
Old Testament	3.3	761,773	0.09 (2.6%)	11,853 (1.5%)	0.09 (2.6%)	12,426 (1.6%)	0.553

Table 2. Vocabulary statistics of the datasets.

Table 3 shows the compression ratios obtained when using multiple containers (distinct vocabularies) and a single container (unique vocabulary).

Dataset	Compression ratio(%)	
	Unique Voc.	Distinct Voc.
XMark	37.08	34.16
DBLP	47.04	43.49
SwissProt	43.75	40.94
Shakespeare	38.32	38.18
MeSH	25.41	21.33
Old Testament	32.87	33.49

Table 3. Comparison between unique and distinct vocabularies.

### 5.2. Compression Ratio

We compared the compression ratios achieved by YAQCX with those achieved by XMill, XGrind and XQzip (see Figure 2). Notice that XQzip is not publicly available, thus our

cross-comparison involved only those datasets considered in [Cheng and Ng 2004]. On the other hand, we did not compare YAQCX with XPRESS and XQueC not only because they are not publicly available, but also because we considered the datasets used in their experiments [Arion et al. 2004, Min et al. 2003] too small, making it difficult a cross-comparison.

The compression ratio  $r = (z/u) \times 100$  denotes the size of the compressed XML document as a percentage of the uncompressed one,  $z$  is the size of the compressed XML document, and  $u$  is the size of the uncompressed one. As we can see from Figure 2, compressed ratios achieved by YAQCX are very close to those achieved by XQzip and, on average, 5% better than those achieved by XGrind. Furthermore, considering the datasets we used, a document compressed by YAQCX takes, on average, about 35% of the size of the original one.

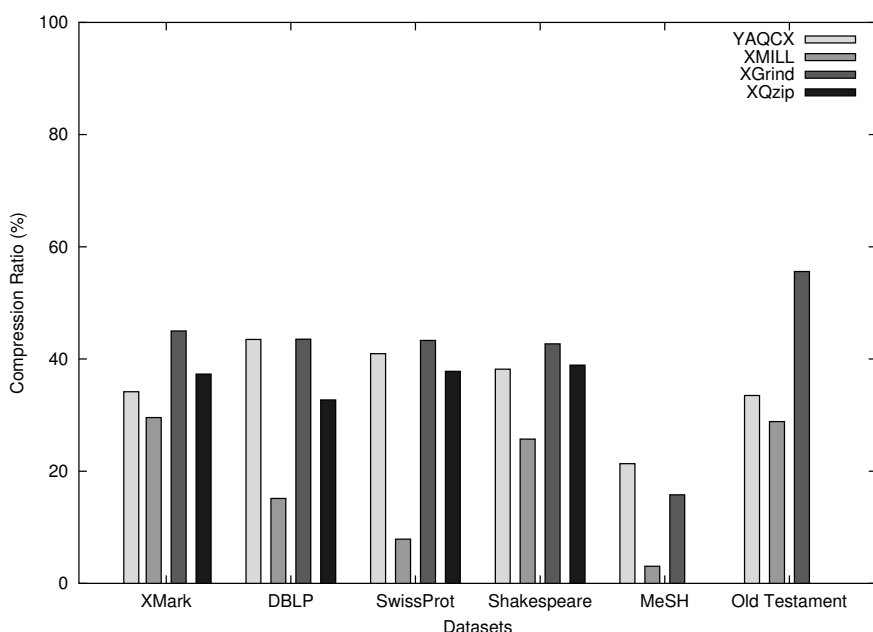


Figure 2. Compression ratio.

### 5.3. Compression/Decompression Time

We compared YAQCX compression/decompression time with XMill and an XGrind version provided by one of its authors, not the one available at SourceForge, that lacks on performance. Figures 3 and 4 summarizes our results for compression/decompression. We notice that YAQCX is, on average, 1.46 times slower than XMill and that XGrind is, on average, 1.32 times slower than YAQCX, meaning that YAQCX is almost the middle point between them considering compression performance. On the other hand, the strongest point of YAQCX is its decompression performance. On average, YAQCX is 4.16 times and 10.36 times faster than XMill and XGrind, respectively. This achievement comes directly from YAQCX coding and modeling, since the decoding algorithm is very simple (no masking operations are needed) and efficient (YAQCX outputs one entire word at once and codes each symbol using bytes).

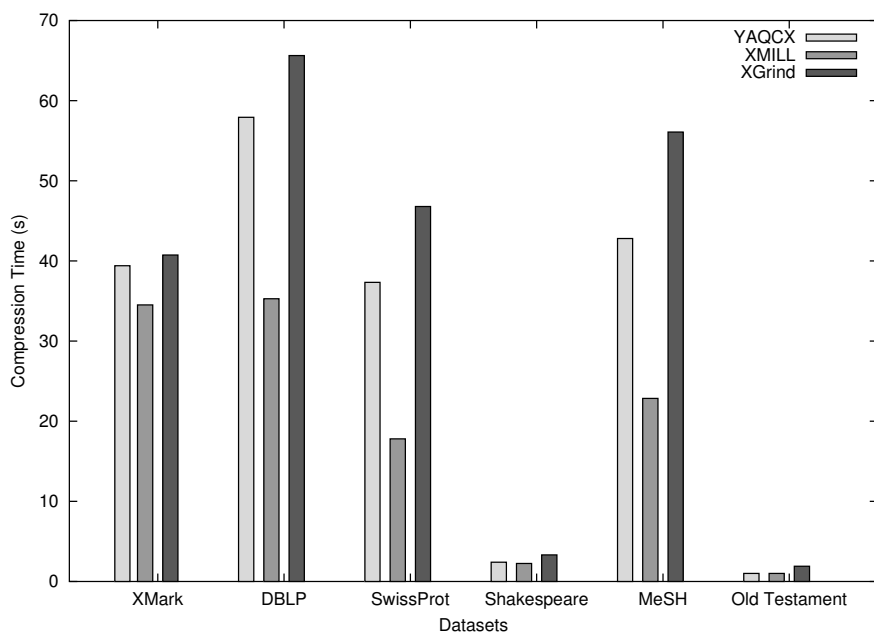


Figure 3. Compression time.

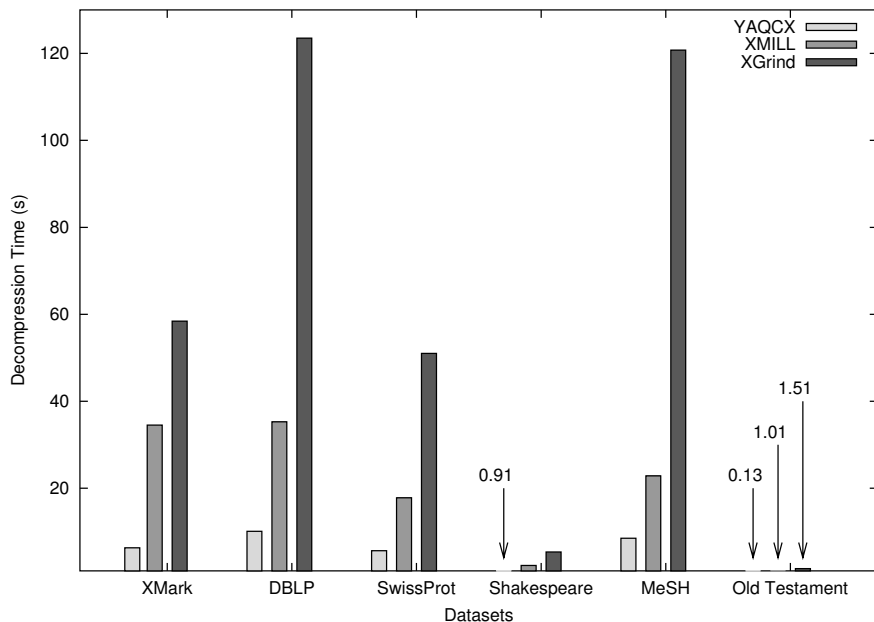


Figure 4. Decompression time.

XMark	SPath	/site/regions/africa/item/mailbox/mail/text
	EMatch	/site/regions/africa/item/mailbox/mail/text[contains("half")]
	RMatch	/site/regions/africa/item/mailbox/mail[date[text()]>="05" and text()<="06"]]/text
	RExpr	/site/regions/africa/item/mailbox/mail/text[contains("ons")]
DBLP	SPath	/dblp/inproceedings/title
	EMatch	/dblp/inproceedings/title[contains("Knowledge Management")]
	RMatch	/dblp/inproceedings[year[text()] >= 2000 and text() <= 2003]]/title
	RExpr	/dblp/inproceedings/title[contains("Lock* Control")]
SwissProt	SPath	/root/Entry/Descr
	EMatch	/root/Entry/Descr[contains("PROTEIN")]
	RMatch	/root/Entry[Org[text()] = "Eukaryota" or text() != "Viridiplantae"]]/Descr
	RExpr	/root/Entry/Descr[contains("PREC*")]
Shakespeare	SPath	/PLAY/ACT/SCENE/SPEECH/LINE
	EMatch	/PLAY/ACT/SCENE/SPEECH/LINE[contains("great fights")]
	RMatch	/PLAY/ACT/SCENE/SPEECH[SPEAKER [text()] >= "LEPIDUS" and text() <= "MECAENAS"]]/LINE
	RExpr	/PLAY/ACT/SCENE/SPEECH/LINE[contains("purpo* about")]
Mesh	SPath	/DescriptorRecordSet/DescriptorRecord/DescriptorName/String
	EMatch	/DescriptorRecordSet/DescriptorRecord/DescriptorName/String[contains("Calcimycin")]
	RMatch	/DescriptorRecordSet/DescriptorRecord[DateCreated/year[text()] >= 1974]]/DescriptorName/String
	RExpr	/DescriptorRecordSet/DescriptorRecord/DescriptorName/String[contains("Calci*")]
Old Testament	SPath	/tstmt/bookcoll/book/chapter/chtitle
	EMatch	/tstmt/bookcoll/book/chapter/chtitle[contains("1")]
	RMatch	/tstmt/bookcoll/book/chapter/chtitle[text()] >= "Chapter 4"]
	RExpr	/tstmt/bookcoll/book/chapter/v[contains("firma* *ven")]

Table 4. Benchmark queries.

#### 5.4. Query Processing Performance

We have tested YAQCX with a number of queries on our six datasets and compared its performance against XGrind. Furthermore, we compared YAQCX performance against Sablotron<sup>3</sup>, a publicly available XPath processor written in C++, in order to better evaluate the improvement achieved as the result of our compression method. We chose Sablotron because it is written in the same language of YAQCX and was the only one able to run large datasets by the time we carried out our experiments.

We devised four types of query with increasing complexity. SPath includes simple path queries with no predicates. EMatch extends SPath with an exact matching predicate in the result nodes. RMatch replaces exact matching by range matching. Finally, RExpr uses arbitrary regular expressions for matching. The complete list of queries is shown in Table 4.

YAQCX processes SPath queries using the structure summary to find out which nodes should be selected and then scans the compressed data to show the results. For EMatch, RMatch and RExpr queries, it uses the structure summary to evaluate the path expression and the compressed data to evaluate the predicates, applying a simple pattern matching algorithm for EMatch queries and a multi-pattern algorithm for RMatch and RExpr queries.

Table 5 shows the results achieved by YAQCX, XGrind and Sablotron in our experiments. However, it is important to notice that we needed to expand the memory of the computer to 1Gb in order to run Sablotron with datasets greater than 100Mb.

From Table 5 we notice that YAQCX is 11.82, 16.34, and 14.65 times faster than XGrind for SPath, EMatch, and RMatch queries. Moreover, processing time

<sup>3</sup><http://www.gingerall.com/charlie/ga/xml/p.sab.xml>

Dataset Name	Query Type	YAQCX Time (s)	XGrind Time (s)	Sablotron Time (s)
XMark	SPath	1.35	15.90	191
	EMatch	1.08	16.19	1765
	RMatch	1.46	24.67	–
	RExp	1.41	–	–
DBLP	SPath	2.03	25.22	1615
	EMatch	1.59	32.84	1789
	RMatch	2.79	60.83	–
	RExp	2.54	–	–
SwissProt	SPath	1.25	14.21	2112
	EMatch	0.90	18.47	1826
	RMatch	2.09	27.13	–
	RExp	1.77	–	–
Shakespeare	SPath	0.19	1.51	10.05
	EMatch	0.16	1.86	7.21
	RMatch	0.34	2.61	–
	RExp	0.30	–	–
MeSH	SPath	1.23	21.50	1884
	EMatch	0.97	25.74	3072
	RMatch	1.39	33.88	–
	RExp	1.32	–	–
Old Testament	SPath	0.14	0.41	1.05
	EMatch	0.13	0.49	0.95
	RMatch	0.20	0.84	–
	RExp	0.31	–	–

Table 5. Query processing results.

of XGrind increases according to the complexity of the query, while for YAQCX it is not always the case. To explain this, we notice that query processing time is spent in two distinct phases: query evaluation and output of results. Since YAQCX uses the structure summary and never decompresses the document to evaluate the query, we can say that the increase of complexity plays a small impact over the processing time. On the other hand, the number of selected nodes decreases from SPath to EMatch queries, thus the time spent outputting results is much smaller. We also notice that, even for arbitrary regular expressions, YAQCX often performs better than when dealing with less complex queries, once again due to the impact of the number of selected nodes.

When comparing to Sablotron, we can easily verify the huge impact of compression over query processing performance. On average, YAQCX is 105.6 and 291.9 times faster than Sablotron for SPath and EMatch queries. Moreover, it is important to notice that the difficulty to deal with large documents is the commonplace for query processors that work on plain XML documents, since they use large in-memory structures.

## 6. Concluding Remarks

In this paper, we have presented YAQCX, a query-aware compressor for XML that is able to efficiently process XPath expressions extended with a powerful pattern matching engine. The greatest strength of YAQCX is its performance on decompression and query processing. The interaction between the structure and contents compression plays an important role here, since the structure summary provides an easy way to find the tags into the compressed data, and byte coding and word-based modeling provides an efficient way to access the compressed contents, with great flexibility.

Analyzing our experimental results, we can say that YAQCX fits better in scenarios in which documents are more static and querying is frequent, thus we compress the

documents only once, with a higher cost, and query them many times directly over the compressed version.

We ponder some ways of improving YAQCX compression ratios, using specific models for distinct containers. For example, depending on the type of data kept in them, YAQCX can choose the compression method that fits better, based on an automatic data type detection algorithm. Notice, however, that improving compression ratios this way may worsen query processing performance, since processing algorithms become more complex due to model mixing.

On the other hand, query processing performance can be improved by adding indexes to YAQCX, that in its turn may worsen compression ratios. Finding the best trade-off between these two points is an interesting open problem that we intend to work on.

### **Acknowledgments**

The authors would like to thank Pankaj Tolani for providing them with a new release of XGrind and James Cheng for his valuable comments regarding XQzip. This work was partially supported by project GERINDO (MCT/CNPq/CT-INFO grant 552087/02).

### **References**

- Arion, A., Bonifati, A., Costa, G., D'Aguanno, S., Manolescu, I., and Pugliese, A. (2004). Efficient Query Evaluation over Compressed XML Data. In *Proceedings of the 9th International Conference on Extending Database Technology*, pages 200–218, Crete, Greece.
- Bell, T. C., Cleary, J. G., and Witten, I. H. (1990). *Text Compression*. Prentice Hall, Upper Saddle, USA.
- Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E. (2000). Extensible Markup Language (XML) 1.0. W3C Recommendation, World Wide Web Consortium. <http://www.w3.org/TR/REC-xml>.
- Buneman, P., Grohe, M., and Koch, C. (2003). Path Queries on Compressed XML. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 141–152, Berlin, Germany.
- Cheney, J. (2001). Compressing XML with Multiplexed Hierarchical PPM Models. In *Proceedings of the 2001 IEEE Data Compression Conference*, pages 163–172, Snowbird, USA.
- Cheng, J. and Ng, W. (2004). XQzip: Querying Compressed XML Using Structural Indexing. In *Proceedings of the 9th International Conference on Extending Database Technology*, pages 219–236, Crete, Greece.
- Clark, J. and DeRose, S. (1999). XML Path Language (XPath) version 1.0. W3C Recommendation, World Wide Web Consortium. <http://www.w3.org/TR/xpath>.
- de Moura, E. S., Navarro, G., Ziviani, N., and Baeza-Yates, R. (2000). Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139.

- Gottlob, G., Koch, C., and Pichler, R. (2002). Efficient Algorithms for Processing XPath Queries. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 95–106. Hong Kong, China.
- Heaps, J. (1978). *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, New York, USA.
- Liefke, H. and Suciu, D. (2000). XMill: an Efficient Compressor for XML Data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, Dallas, USA.
- Lin, Y., Zhang, Y., Li, Q., and Yang, J. (2005). Supporting efficient query processing on compressed xml files. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 660–665, Santa Fe, USA.
- Min, J., Park, M., and Chung, C. (2003). XPRESS: A Queriable Compression for XML Data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 122–133. San Diego, California.
- Ng, W., Lam, W., and Cheng, J. (2006). Comparative analysis of xml compression techniques. *World Wide Web: Internet and Web Informations Systems*, 9:5–33.
- Schmidt, A. R., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., and Busse, R. (2002). XMark: A Benchmark for XML Data Management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 974–985. Hong Kong, China.
- Tolani, P. and Haritsa, J. R. (2002). XGRIND: A Query-friendly XML Compressor. In *Proceedings of the 18th International Conference on Data Engineering*, pages 225–234. San Jose, USA.