Comparative Analysis of XML Compression Technologies

Wilfred Ng   Lam Wai Yeung   James Cheng
Department of Computer Science
The Hong Kong University of Science and Technology
Hong Kong
Email: {wilfred, yeung, csjames}@cs.ust.hk

Abstract

XML provides flexibility in publishing and exchanging heterogeneous data on the Web. However, the language is by nature verbose and thus XML documents are usually larger in size than other specifications containing the same data content. It is natural to expect that the data size will continue to grow as XML data proliferates on the Web. The *size problem* of XML documents hinders the applications of XML, since it substantially increases the costs of storing, processing and exchanging the data. The hindrance is more apparent in bandwidth- and memory-limited settings such as those applications related to mobile communication.

In this paper, we survey a range of recently proposed XML specific compression technologies and study their efforts and capabilities to overcome the size problem. First, by categorizing XML compression technologies into queriable and unqueriable compressors, we explain the efforts in the representative technologies that aim at utilizing the exposed structure information from the input XML documents. Second, we discuss the importance of queriable XML compressors and assess whether the compressed XML documents generated from these technologies are able to support direct querying on XML data. Finally, we present a comparative analysis of the state-of-the-art XML conscious compression technologies in terms of compression ratio, compression and decompression times, memory consumption, and query performance.

1 Introduction

Extensible Markup Language (XML) [19] is proposed as a standardized data format designed for specifying and exchanging data on the Web. With the proliferation of mobile devices, such as palmtop computers, as a means of communication in recent years, it is reasonable to expect that in the foreseeable future, a massive amount of XML data will be generated and exchanged between applications in order to perform dynamic computations over the Web. However, XML is by nature verbose, since terseness in XML markup is not considered a pressing issue from the design perspective [19]. In practice, XML documents are usually large in size as they often contain much redundant data, such as repeated tags (c.f. DBLP documents in [17]). The *size problem* hinders the adoption of XML, since it substantially increases the costs of data processing, data storage, and data exchanges over the Web.

As the common generic text compressors, such as Gzip [20], Bzip2 [44], WinZip [52], PKZIP [37], or MPEG-7(BiM) [34], are not able to produce usable XML compressed data, many XML specific compression technologies have been recently proposed. The essential idea of these technologies is that, by utilizing the exposed structure information in the input XML document during the compression process, they pursue two important goals at the same time. First, they aim at achieving a good compression ratio and time compared to the generic text compressors mentioned above. Second, they aim at generating a compressed XML document that is able to support efficient evaluation of queries over the data.

In our survey of XML-conscious compressors we find that the existing technologies indeed trade between these two goals. For example, XMill [30] needs to perform a *full* decompression prior to processing queries over compressed documents, resulting in a heavy burden on system resources such as CPU processing time and memory consumption. At the other extreme, some technologies can avoid XML data decompression in some cases, but unfortunately only at the expense of the compression performance. For example, XGrind [46] adopts a homomorphic transformation strategy to transform XML data into a specialized compressed format and support direct querying on compressed data, but only at the expense of the compression ratio; thus the XML size problem is not satisfactorily resolved.

In regard to the importance of achieving a good level of performance in both compression and querying, we find that the current research work on XML compression does not adequately analyze the related features. We classify the existing XML-conscious XML compression into the categories of *unqueriable compression* and *queriable compression.* We present a qualitative analysis of eight representative XML-conscious compression technologies as follows. The unqueriable compression includes *XMill* [30], *XMLPPM*, [12] *SCA* [29], and *Millau* [42]. The queriable compression includes *XGrind* [46], *XMLZip* [53], *XPress* [35] and *XML Skeleton Compression* [7]. All of these compression technologies are known to the community, which contains interesting features and important contributions related to XML compression. We present a critical comparison of these technologies.

Specifically, we make the following contributions:

- We present a comprehensive survey of the state-of-the-art XML compressors and analyze their important features, relative strengths, and limitations.
- We evaluate the performance of the compression technologies on a range of important XML benchmark data. The performance indicators include compression ratio, compression time, memory usage, and query support issues.
- We propose a set of desirable qualities for XML compressors and discuss some further research areas, that we believe are important for the further development of XML conscious compression.

The paper is organized as follows. In Section 2, we introduce four unqueriable compression technologies that are optimized for achieving better compression. Then, in Section 3, we introduce another four queriable XML-conscious compression technologies that are able to support querying over compressed XML documents. Some recently proposed compressors during the time of writing are also discussed. In Section 4, we present a comprehensive comparison of six XML conscious compression technologies and evaluate them in terms of compression ratio, compression and decompression times, memory usage, and query support. In Section 5, we propose a set of desirable features of an XML compressor and discuss the challenges. In Section 6, we discuss some considerations when deploying the compression techniques. Finally, we make concluding remarks and highlight possible areas for further research in Section 7.

2 Unqueriable XML Compression

We now review four compressors that show good compression performance but do not support querying over the compressed data. We assume that the readers have basic knowledge of XML and its languages [19, 18, 14, 5]. Throughout this paper, we use the XML publication catalog snippet shown in Figures 1(a) and 1(b) as a running example for illustrating the features in various compression technologies.

```
<paper>
  <entry year="2003">
    <journal>
      <title>Secret Sharing</title>
    </journal>
  </entry>
  <entry year="2003">
    <conference>
      <title>XML Water Mark</title>
    </conference>
  </entry>
</paper>
```

(a) A Publication Catalog XML Snippet

```
<!ELEMENT paper (entry*)>
<!ELEMENT entry (journal|conference)>
<!ATTLIST entry year CDATA>
<!ELEMENT journal (title)>
<!ELEMENT conference (title)>
<!ELEMENT title (#PCDATA)>
```

(b) A Publication Catalog DTD

Figure 1: A Running Example of an XML Publication Snippet and Its Corresponding DTD

2.1 XMill

XMill [30] is the first proposed XML-conscious compression architecture in the literature, that separates the structural information from data of the input document during the compression process. The *structural items* include element tag names, attribute names, and the document skeleton generated in a compact form. The *data items* include PCDATA and attribute values. Prior to the main compression process, a pre-compression phase is introduced in XMill before the data is compressed by Gzip. There are two specific objectives in the pre-compression phase in order to optimize XML compression:

1. to separate the document structural information from data, and
2. to group data items with related semantics in the same "container".

First, in order to extract the structural item from the documents, XMill adopts a *dictionary encoding approach* [4] to store the tag and attribute names in a dictionary. A corresponding compact document skeleton is then generated by replacing element tags and attribute names with dictionary indexes. This compact skeleton is then stored in a *structure container*, as shown in Figure 2, where T_i, A_i and and "/" are the code for a start tag, an attribute and a closed tag, respectively. Second, in order to group data items for efficient compression, XMill uses an *approximation matching* on the *Reversed DataGuide* [23, 30] for determining which containers the data values belong to. In its default mode, data items having the same tag or attribute name are grouped into the same container. On the other hand, in its intervention mode, it allows users to determine the method of data grouping. In this mode, *container expressions* can be specified as command line parameters to instruct the XMill compressor how to group data items, and then specific *semantic compressors* can be employed in the path processor to perform semantic compression over the corresponding data container. This further helps to achieve a compression ratio that is up to 50% better than when running data grouping in a default mode. However, in the intervention mode user expertise and effort are required to manually intervene in the compression process in order to achieve highly specialized compression such as DNA sequences.

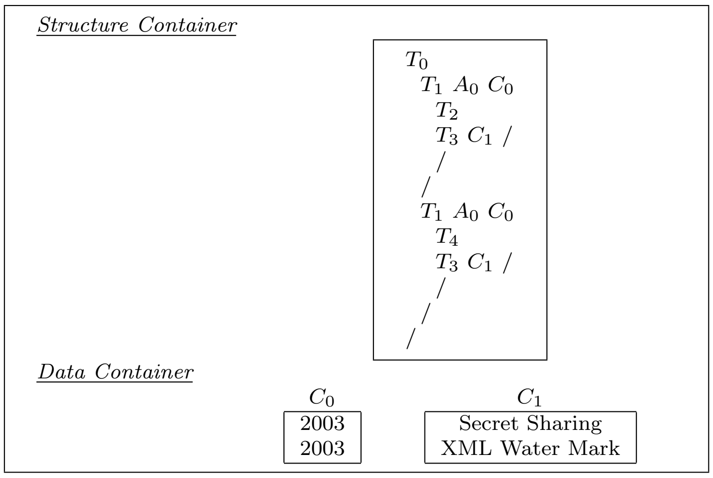


Figure 2: A Conceptual View of the XMill Compressed Publication Snippet

Figure 2 depicts a conceptual view of the XMill compressed document of the XML publication snippet shown in Figure 1(a). XMill does not use the DTD of an XML document in its compression. Each data container in XMill is then compressed individually in the main compression phase by using Gzip, whose output is concatenated in a single file. Overall, XMill is an efficient XML compressor due to the fact that it is optimized for achieving a good compression ratio for XML documents. However, it needs to perform decompression of the whole of the XML documents before the evaluation of the imposed queries over the compressed documents. The intervention mode also adds some burden to the users.

2.2 XMLPPM

XMLPPM [12] proposes a technique called Multiplexed Hierarchical Modeling (MHM), which is based on SAX encoding [38] and Prediction by Partial Match (PPM) encoding [16]. In XMLPPM, an input XML document is modelled as a stream of SAX events. The tokens in the SAX event stream are then processed by a set of four PPM coding models corresponding to different syntactic contexts as follows:

1. the *element and attribute name model* (*Syms*),
2. the *element structure model* (*Elts*),
3. the *attribute values model* (*Atts*), and
4. the *string value model* (*Chars*).

Depending on the syntactic contexts, the SAX tokens are sent to their corresponding PPM models for running predictions and encodings. Figure 3 shows the status of the four PPM models when the XML publication catalog snippet is compressed by XMLPPM. First, the XML document is converted into a corresponding stream of SAX events. The tokens in the SAX event stream are then sent to their corresponding PPM models for encoding based on the syntactic contexts of the tokens. For example, when the *paper* start element token is encountered, the string value *paper* is sent to the *Syms* model, since this element name was not encoded before. Moreover, a byte value *01* is assigned to this string value, which is used to represent the string when it is encountered again. This tokenization process is only applied to the element tag names and attribute names. Then the token *01* is sent to the *Elts* model, which represents that a *paper* start element token is encountered.

Attribute values are sent to the *Atts* model for encoding. For example, when the attribute value *"2003"* is encountered, its corresponding attribute name token is *0A*, and the values are sent to the *Atts* model. When a data value, such as *"Secret Sharing"*, is encountered, a token *FE* is sent to the *Elts* model and the data value is sent to the *Chars* model for encoding. Note that all the end tags are replaced by the token *FF*. As an element tag can be strongly correlated with its enclosed elements and data, XMLPPM also *injects* the enclosing token index (i.e. the token indexes of the format $\langle nn \rangle$) into the corresponding Elts, Atts, or Chars model immediately before an element, an attribute, or a data value is encoded in order to retain such cross-model dependencies among the tokens in different contexts. However, these injected token indexes indicate to the models that a particular token has been seen, but these token indexes are not explicitly encoded in the models.

The enclosing token index injection has two benefits. First, it prevents that the separation of event tokens breaks the *cross-model dependencies* among the tokens. The happening of cross-model dependencies is due to the fact that neighboring symbols from different syntactic classes may not be drawn from distinct independent sources. For example, the cross-model dependency occurs when the enclosing element tag is strongly correlated with enclosed data. This phenomenon is common in most XML data and thus XMill introduces the data container to group data items under the same tag element (recall Figure 2 in Section 2.1). As the symbols in different syntactic classes are likely to have a strong correlation, multiplexing the four mentioned models may break existing cross-class sequential dependencies and thus XMLPPM is not able to make an accurate estimation of probability range that are used to transmit symbols using arithmetic coding. An injected symbol gives the information to the model and helps the PPM models to retain and utilize the cross-model dependencies without sacrificing storage space. Second, the token index can also be utilized for improving the compression efficiency, since tokens from different syntactic contexts in the neighborhood can be strongly correlated to each other. For example, the enclosing token indexes ⟨02⟩ and ⟨04⟩, which represent the enclosing tags *entry* and *title*, respectively, are injected into the *Atts* and *Chars* models immediately before an element, an attribute, or a data value is encoded.

(1)	⟨paper⟩	⟨entry	year=	"2003"	⟩	⟨journal⟩	⟨title⟩	Secret Sharing
Syms:	paper 00	entry 00	year 00			journal 00	title 00	
Elts:	01	02				03	04	FE
Atts:			⟨02⟩0A	2003 00	⟨02⟩FF			
Chars:								⟨04⟩Secret Sharing 00

(2)	⟨/title⟩	⟨/journal⟩	⟨/entry⟩	⟨entry	year=	"2003"	⟩	⟨conference⟩
Syms:								conference 00
Elts:	FF	FF	FF	02				05
Atts:					⟨02⟩ 0A	2003 00	⟨02⟩ FF	
Chars:								

(3)	⟨title⟩	XML Water Mark	⟨/title⟩	⟨/conference⟩	⟨/entry⟩	⟨/paper⟩
Syms:						
Elts:	04	FE	FF	FF	FF	FF
Atts:						
Chars:		⟨04⟩XML Water Mark 00				

Figure 3: Multiplexed Hierarchical Modeling in XMLPPM

Compared to XMill, XMLPPM has the benefit of supporting the on-line processing of compressed documents (Encoded SAX). Importantly, XMLPPM does not rely on user intervention but is still able to achieve a better compression ratio than that of XMill (default mode). The underlying reason is that PPM is a finite context statistical modelling technique, which can be viewed as blending together several fixed-order context models to predict the next character in an incoming SAX token sequence. Prediction probabilities for each context in the model are calculated from frequency counts that are updated adaptively. The symbol that actually occurs is encoded relative to its predicted distribution using arithmetic coding. However, XMLPPM requires a longer compression time than others, since PPM is known to be a relatively slow compression technology (c.f. see Table 2 in [16]).

2.3 Millau

Millau [42] is a system designed for efficient encoding and streaming of XML documents. This system comprises a set of compression and encoding techniques that are dedicated for XML compression. The principle technique used in Millau is called the *Differential DTD Tree Compression* (DDT), which is an XML-conscious compression technique that makes use of the information from DTDs to facilitate better compression. The main idea behind DDT is that it encodes only the document information that cannot be inferred from the document associated DTD, which includes *data values* and *structural information* of the XML document. The data values here refer to the PCDATAs and attribute values. The structural information here refers to the values of the operators in the DTD, such as optional operators "?", decision operators "|", and repetition operators "∗" and "+". The experimental results reported in [42] show that this compression strategy yields a good compression ratio.

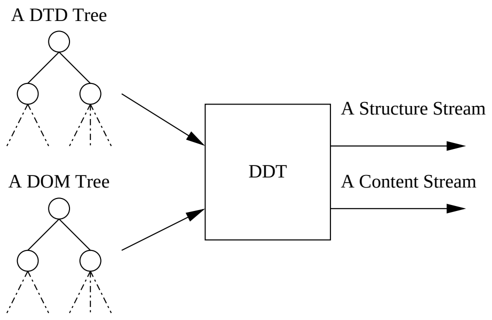


Figure 4: The DDT Compression Approach

DDT compression only handles valid XML documents that conform to a given DTD. Figure 4 depicts the underlying ideas used in this approach. Both the input DTD and the XML document are represented as trees in the compression process. A special tree parsing algorithm is used to parse the DTD tree and the DOM tree simultaneously where the processing nodes in the trees possess the same semantics. This parsing process is used to explore the structural information of the input XML document, which is outputted to a structure stream. Data values of the input XML document are also extracted and outputted to a content stream, which is then compressed in the main compression phase.

```
Structure Stream: 2 0b 1b
Data Stream: 2003|Secret Sharing|2003|XML Water Mark
```

Figure 5: A Conceptual View of the Millau Compressed Publication Snippet

We show in Figure 5 the Millau streams for the XML document shown in Figure 1(a). In the structure stream of this figure, the integer "2" is assigned to the first occurrence of the repetition operator "∗". The assignment represents the fact that there are two "entry" nodes in the DOM tree. The byte "0b" is assigned to the first occurrence of the decision operator "|" in the DTD tree. The assignment represents the fact that the child node of the first “entry” node is a “journal” node. Similarly, the assignment of the byte “1b” represents the fact that the child node of the first “entry” node is a “conference” node. The data stream is extracted from the document and the values of the stream are trivial.

It is worth mentioning that an interesting contribution of XML compression from Millau is on the interoperablility side. The work in [42] presents two applications on top of the Millau APIs related to e-Business transaction on the Internet. The first one is a compression-decompression proxy server which takes advantage of the compact representation of XML that Millau provides to save Internet network bandwidth and of the ease of processing. The second one uses the methods which return token instead of strings for faster processing of parameters marshaled in XML.

Although DDT is able to achieve a good compression ratio, we find that it suffers from the problem of *huge memory consumption* when compressing large XML documents. This means that in this approach, when parsing an XML document, in order to create a corresponding tree structure, a large portion of memory is required for storing the generated DOM tree. The vigorous use of virtual memory, in practice, results in extremely frequent thrashing of disk I/O [41], which seriously degrades the efficiency of the compression process.

2.4 Structure Compression Algorithm (SCA)

Structure Compression Algorithm (SCA) [29] is an XML-conscious compression algorithm that aims at compressing the structure (skeleton) of an XML document. SCA only handles *valid* XML documents that conform to DTDs. Similar to Millau DDT described in Section 2.3, when compressing an XML document, SCA utilizes information in the associated DTD as *hints* in the compression process, and only encodes the document structure information that cannot be inferred from the given DTD, which also includes *data values* and *structural information* of the XML document. The difference between SCA and DDT is that SCA uses different data structures and parsing approaches to process the input DTD and XML document. It is also formally shown in [29] that, with the assumption that all operator nodes are encoded independently, this compression strategy achieves *optimal* encoding of the skeleton of an XML document.

In SCA, a special tree structure is generated for the input DTD and the XML document collectively. This created tree is called a *Parse Tree* (PaTree). Figure 6 depicts the PaTree created based on the publication catalog XML snippet and its DTD. We can see that this created PaTree is very similar to the well-known DOM tree representation of the XML document. However, an essential difference between them is that there are operator nodes, such as repetition nodes “$*$” and decision nodes “|”, inserted into the PaTree. The non-leaf nodes of the PaTree describe the structure of the XML document, whereas the leaf nodes describe the textual data of the document. This PaTree is then processed by using a *pruning process*, which collapses the tree to form a *Pruned Tree* (PuTree).

Figure 7 depicts the PuTree that corresponds to the PaTree shown in Figure 6. This PuTree maintains only the document structure nodes that are not able to

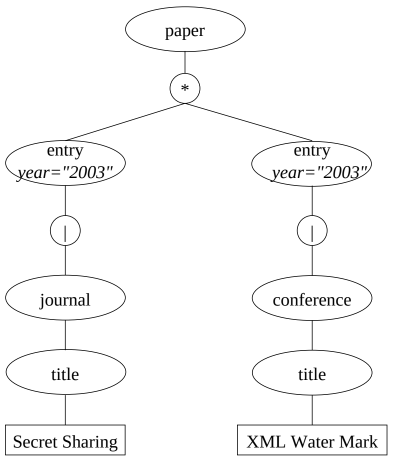


Figure 6: The Parse Tree for the XML Publication Snippet

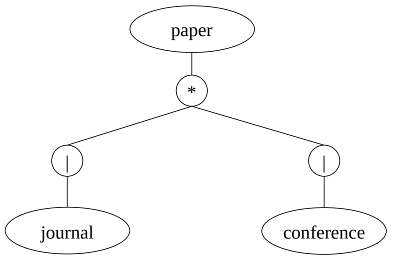


Figure 7: The Pruned Parse Tree for the XML Publication Snippet

be inferred from the given DTD, which are needed to be encoded during the structure encoding process. Data values of the input XML document are extracted and outputted to a content stream, which is then compressed by a generic compressor, such as Gzip, in the main compression phase. After the PuTree is created, the SCA coder starts the document structure encoding process by using a *breadth-first traversal* [50] to navigate the PuTree in order to explore the required structural information for generating the output encoding. However, we find that SCA also suffers from a similar problem of *huge memory consumption* with the DDT technology when compressing a large XML document, since a large portion of memory is required for storing the corresponding generated PaTree and PuTree during the compression process.

3 Queriable XML Compression

In this section, we review four XML compressors that are able to generate queriable compressed XML documents. We also discuss the very recent proposals (up to the year 2004) that all support querying over compressed XML data.

3.1 XGrind

XGrind [46] adopts a *homomorphic transformation* [24] strategy to transform an XML document into an XGrind compressed form. The main advantage of using the transformation is that the generated compressed document preserves the syntactic structure and semantics information of the original XML document. This implies that the compressed document can be parsed in the same way as with any other XML document using the same SAX or DOM parser, without performing decompression.

```
$T_0$
  $T_1$ $A_0$ $\mathcal{H}$(2003)
    $T_2$
    $T_3$ $\mathcal{H}$(Secret Sharing) /
    /
  /
  $T_1$ $A_0$ $\mathcal{H}$(2003)
    $T_4$
    $T_3$ $\mathcal{H}$(XML Water Mark) /
    /
  /
/
```

Figure 8: The Conceptual View of the XGrind Compressed Publication Snippet

Figure 8 depicts the conceptual view of the XGrind compressed XML publication snippet. The notations are similar to the convention used in Section 2 as follows: T_n encodes a start tag, "/" encodes a closed tag, A_n encodes an attribute, and $\mathcal{H}(a)$ represents the Huffman code [25] of a data value a. XGrind relies on the DTD of a given document to populate a symbol table for attributes having enumerated-type values such that they can be encoded differently from other text values, and employed to validate the compressed document. The DTD is also used to initialize the frequency tables for the elements and non-enumerated attributes.

An XGrind compressed document is generated as follows. First, XGrind creates a dictionary for storing all the element and attribute names appearing in the DTD, which will be used for dictionary encoding of the element tags and attribute names of the input document in the compression phase. The compressor then parses the input XML document once (first parse) in order to collect the statistics for the PCDATAs and non-enumerated attribute values. These statistics are used to create coding models for a set of *non-adaptive context-free* Huffman coders for compressing the PCDATAs and non-enumerated attribute values.

After the Huffman coders are created, the compressor initiates a compression process that requires to parsing the XML document again (second parse). The compressor then tokenizes all the tag and attribute names in the XML document by using indexes that point to their corresponding entries in the created dictionary. Enumerated attribute values are binary encoded based on their corresponding symbol tables. PCDATAs and general attribute values are compressed *individually* by using their corresponding Huffman coders. After completing the compression process, the compressed document and all the meta-tables (i.e. the dictionary, the symbol tables of the enumerated-type attributes, and the statistic tables that are used in the Huffman coders) are packed as outputs.

As the output compressed documents in XGrind are homomorphic transformations of their corresponding input documents, all operations that can be executed over the original documents, such as querying, are preserved. These operations can be executed using existing techniques and tools with some modifications. In addition, queries requiring an *exact match* on data values can be executed without decompressing the document. This is because the query processor can transform the data values in the query to their corresponding Huffman encoded values for comparing during query processing. However, the compression ratio delivered by XGrind is in fact much worse than that of XMill. In our experiment, it is found that XGrind compresses an 89 MB Weblog XML document into a 38 MB compressed document, while XMill is able to compress the same document to only 2.3 MB. Also, XGrind compression is much longer than that of XMill, since it requires parsing the input XML document twice during the compression process.

3.2 XML Skeleton Compression

Buneman et al. propose a framework that allows path queries to be evaluated directly on compressed XML documents [7]. The focus of the paper is different from XGrind, in that the skeleton compression aims at reducing the complexity and redundancy of the document structure, rather than the textual data items in the document. In their framework, an XML document is modeled as a tree-like structure. The compressor first separates the document's structural information, which they refer to the *skeleton* of the document, from textual data, as shown in Figure 9(a). It then compresses the extracted skeleton by using a technique based on the idea of *sharing common subtrees*, which transforms the skeleton into a directed acyclic graph (DAG) [50], as shown in Figure 9(b).

The DAG in Figure 9(b) can be further compressed by replacing any consecutive sequence of outgoing edges to the same vertex using a single edge marked with the appropriate cardinality (i.e. two paper/entry links), as shown in Figure 9(c). This compressed skeleton is much smaller than its non-compressed version. Thus, the compressed skeleton can be stored in the main memory, even for a large XML document. It is also formally shown that the skeleton compression algorithm is able to generate an *equivalent* compressed instance that is minimal in terms of the number of vertices.

Although the technique is able to compress the structure of an XML document well, the overall compression ratio (including textual data) achieved by this framework, as mentioned in [7], is worse than that of XMill. However, using the proposed compression technique, the authors formally study the accessing techniques for Core XPath, which is a simplified version of XPath. A core XPath query is evaluated by manipulating the compressed skeleton instance with only partial decompression. The technique allows the navigational aspect of query evaluation, which is responsible for a large portion of the query processing time, to be carried out in the main memory. Thus, the query evaluation is extremely efficient. The complexity of query evaluation is found to be $O(2^{|Q|} \times \mid I \mid)$, where Q is the query size and I is the skele-

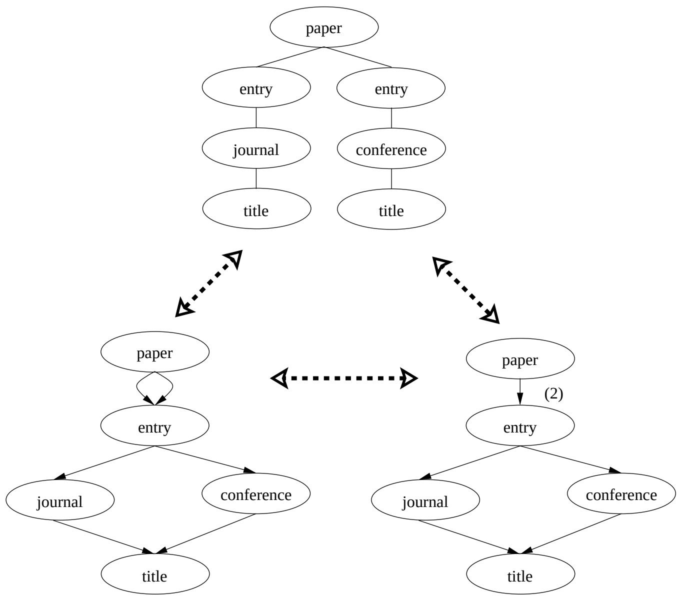


Figure 9: (a) The XML Skeleton (top), (b) the Compressed Skeleton (left), and (c) the Minimal Compressed Skeleton for the XML Publication Snippet (right)

ton instance size. The exponential factor arises from the decompression cost. If no decompression is needed for the evaluation, the complexity reduces to $O(| Q | \times | I |)$.

3.3 XPress

XPress [35] proposes an XML compressor that supports direct querying over compressed XML documents. Similar to XGrind, XPress adopts a homomorphic transformation strategy to transform an XML document into a compressed form, that preserves the syntactic and semantic information of the original XML document.

```
I(/paper)
  I(/paper/entry) I(/paper/entry/@year) H(2003)
    I(/paper/entry/journal)
    I(/paper/entry/journal/title) H(Secret Sharing) 0x80
    0x80
  0x80
  I(/paper/entry) I(/paper/entry/@year) H(2003)
    I(/paper/entry/conference)
    I(/paper/entry/journal/title) H(XML Water Mark) 0x80
    0x80
  0x80
0x80
```

Figure 10: A Conceptual View of the XPress Compressed Publication Catalog Snippet

XPress is built based on a novel coding scheme called *Reverse Arithmetic Encoding.* This coding technique is designed for encoding the *tree paths* of the elements in an XML document using *real number intervals*. Each of these real number intervals falls in the range from 0.0 (inclusive) to 1.0 (exclusive), which represents their corresponding root node to element node path. These number intervals, which represent the encoded element paths, have an important feature that they conform to the *suffix containment* property. This property ensures that if an element path P is a *suffix* of an element path Q, then the interval that represents P, denoted as I_p, should contain the interval that represents Q, denoted as I_q. For example, if the element paths in our XML snippet example are encoded using Reverse Arithmetic Encoding, the interval that represents the element path "//journal/title/" contains the interval that represents the element path "/paper/entry/journal/title" since "//journal/title/" is a suffix of "/paper/entry/journal/title".

PCDATAs and attribute values in the document are compressed *individually* using different *context-free compression methods* depending on their data types based on the following three scenarios:

1. If the data values under a particular tree path belong to a *numerical domain*, the compressor transforms the data values into their corresponding *binary representations* and applies *differential encoding* on the transformed values.

2. If the data values under a particular tree path are *enumerated-type values*, the compressor encodes them using *dictionary encoding*.

3. If the data values under a particular tree path are *string values*, the compressor encodes them using *context-free Huffman encoding*.

The coding scheme in XPress improves the compression strategy adopted in XGrind in two aspects. First, instead of just encoding the element tags of the elements in an XML document, XPress encodes the *tree paths* of the elements using real number intervals, that satisfy the suffixes containment property. As such, XPress is able to evaluate the path-based queries over compressed XML documents directly by checking the interval containment between the paths in an imposed query and the paths of the elements in the compressed document without decompression. Second, numerical domain data values in XPress compressed documents are encoded using *order preserving context-free compression* methods. This allows XPress to evaluate *exact match* and *range* queries concerning numerical data over compressed documents directly without decompressing the data values.

However, as also mentioned in [35], the compression ratio of XPress is in fact worse than that of XMill. In addition, XPress compression time is almost twice as long as that of XMill, since it requires parsing the input XML document twice in the compression process.

3.4 XMLZip

XMLZip [53] compresses XML documents that are represented as DOM trees. Like XMill, XMLZip is built on the generic text compressor Gzip. XMLZip essentially parses and divides an XML DOM tree at a certain pruning depth into multiple components, and then compresses the components separately.

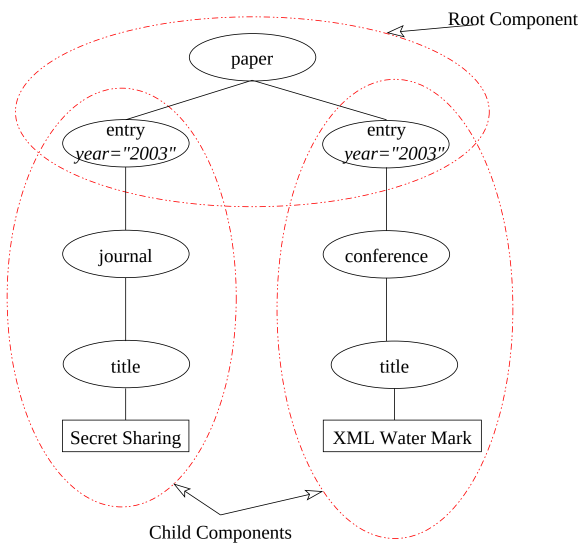


Figure 11: DOM Tree Component Separation in XMLZip

As illustrated in Figure 11, the DOM tree of the publication snippet is divided into a *root component*, which contains all the nodes in the tree up to depth d from the root, and child components for each of the remaining subtrees starting at depth d. These components are then compressed individually by Gzip. Users are allowed to change the pruning depth d, when selecting the DOM level at which to pack the data.

In fact, the node level grouping strategy in XMLZip does not improve compression efficiency. We find that the compression ratio achieved by XMLZip is usually worse than that achieved by Gzip, which will be detailed in Section 4.2.1. However, the advantage of XMLZip is that it allows limited random access to partially decompressed XML documents, since XMLZip supports decompressing the portions of the compressed components that are needed in query evaluation.

3.5 Recent Proposals

At the time of writing (2003-2004), there are few emerging XML compression technologies. It is worth mentioning the following very recently proposed XML compressors: XQueC [3, 2], XQzip [13], and XCQ [28]. All three of them claim to be able to support the querying of compressed XML data to different extents and to maintain a reasonably good compression performance in the meantime.

Like XGrind and XPRESS, XQueC compresses individual data items of an XML document in order to avoid decompression during query evaluation. However, it differs from XGrind and XPRESS in that it separates the XML structure from the XML data items. Data items specified by the same root-to-leaf path are grouped into the same container. XQueC can choose to compress the XML data by applying either the ALM algorithm [1] to preserve order in the encoded data for efficient evaluation of inequality predicates or the classical Huffman algorithm [25] to support prefix-wildcards (but not inequality). A cost model is devised to perform the choice of a suitable compression algorithm and to group the compressed data granules according to their common properties, based on the given query workload. XQueC supports a large subset of XQuery [5] queries. To support efficient query processing, XQueC constructs the structure tree of the input XML document and its structure summary that represents all distinct paths in the document. To allow efficient access of the compressed data items, XQueC links each individually compressed data item to its corresponding node in the structure tree and links each container to the corresponding path in the structure summary. Although XQueC achieves significantly improved query performance and query expressiveness over XGrind and XPRESS, the fine-grained compression adopted in XQueC is an insufficient use of the commonality of the XML data and hence results in a worse compression ratio than XMill. Moreover, the auxiliary data structures, in particular the structure tree and the pointers to and from the individually compressed data items, might incur huge space overhead.

XQzip supports a wide spectrum of XPath queries on compressed XML data by introducing a Structural Indexing Tree (SIT) scheme. The SIT, which evolves from the notion of covering indexes [27], is constructed based on the partitioning of equivalent paths in an XML document. The data are separated from the structures (i.e., the SIT) and compressed into distinct data containers, which are further divided into smaller data blocks that can be compressed and decompressed as an individual unit. This block-compression exploits a trade-off between the compression ratio and decompression overhead incurred in query processing: it takes advantage of the commonality of the XML data to achieve a good compression ratio and at the same time avoids full decompression in query evaluation. The data blocks also lend themselves naturally to the employment of a buffer pool, which avoids repeated decompression in query evaluation if the data is already in the pool. With the use of the SIT index and the buffer pool, XQzip achieves a competitive querying time, especially when the workload of queries is high and exhibits a high degree of locality. However, since the characteristics of input XML documents may vary greatly, a suitable block size is difficult to find and hence a good balance between the compression ratio and the stability of the query performance may be difficult to maintain for some datasets. Moreover, the SIT index does not support the evaluation of complex queries such as joins and order-based predicates.

XCQ is a schema-aware XML-conscious compressor. The compressor supports the evaluation of a subset of XPath queries as well as aggregation queries over compressed XML data by partial decompression. By exploiting the structural information in the input document and its associated DTD, XCQ restructures the input document into distinct data streams in a path-based manner. These data streams are then partitioned into indexed blocks that can be compressed and decompressed as an individual unit. Since the compressed contexts are confined to a single compressed data block, only those blocks that contain information relevant to the query are needed to be decompressed when evaluating an imposed query. However, one of the limitations of XCQ is that it only processes valid XML documents (i.e. documents conforming to a given DTD). It also requires longer compression and decompression times when forming the XCQ partitioned data streams.

4 Comparative Analysis of Performance

In the previous sections, we introduced a collection of currently known XML-conscious compression technologies and outlined their underlying working principles. We also briefly discussed their relative strengths and weaknesses. In this section, we compare the compression performance and query performance of some of the compression technologies discussed above. We exclude skeleton compression [7], since it compresses against XML structures only and is not directly comparable with the others. We also exclude XMLZip [53] in the discussion of query coverage, since it does not have a query processor.

4.1 XML Datasets

We use six data sources for the experiments that cover a wide range of XML data formats and structures. Some characteristics of these data sources are shown in Table 1, where E_num and A_num refer to the number of elements and attributes in the document, respectively. We also show in the last column the parameter *pruning depth* of XMLZip we set for different datasets. The following is the description of each data source: XMark, produced by the *xmlgen* of the XML benchmark project XMark [39], models an auction database with deeply nested elements. DBLP [17], the popular bibliography database, is relatively regular. SwissProt [43], which describes the DNA sequences, has a minimal level of redundancy. Shakespeare [6] is a corpus of marked-up Shakespeare plays that contains many long textual passages. TPC-H is the TPC-H benchmark database [47] converted into XML format. Weblog [31], which is a set of real world web-server log files converted into XML format, has a relatively regular structure.

Table 1: XML Datasets for Comparison Experiments

Dataset	File Size	Depth	E_num	A_num	XMLZip Depth
XMark	97MB	4	2873293	621490	2
DBLP	42MB	3	1107711	118028	2
Shakespeare	7.8MB	5	179072	0	5 (no partitioning)
TPC-H	34MB	3	1022976	0	2
Weblog	30MB	3	641037	0	2
SwissProt	21MB	4	618412	336073	2

4.2 Compression Performance

Using an extensive set of experiments, we compare the compression performance of the following compression technologies: XMill, XGrind, XMLPPM, and XMLZip, along with GZip as a reference. We have not empirically compared those systems that have not released a version of their systems at the time of writing. All experiments were run on a Windows 2000 machine with a Pentium III, 600 MHz processor and 192 MB of main memory. We measure the following three performance metrics for five compressors: (1) the compression ratio; (2) the compression and decompression times; and (3) the memory usage.

4.2.1 Compression Ratio

There are two different expressions that are commonly used to define the *Compression Ratio* (CR) of a compressed XML document (c.f. see the different CR definitions used in [12, 30, 35, 46]). The first compression ratio, denoted as CR_1, is expressed as the *number of bits required to represent a byte.* Using CR_1 a better performed compressor achieves a relatively *lower* value. On the other hand, the second compression ratio, denoted as CR_2, is expressed as the *fraction of the input document eliminated.* Using CR_2, a better performed compressor achieves a relatively *higher* value. The essential difference between the two ratios is that CR_1 is proportional to the size of the space that is actually used to store the compressed document, while CR_2 only shows the percentage reduction of the compressed file size to the original file size.

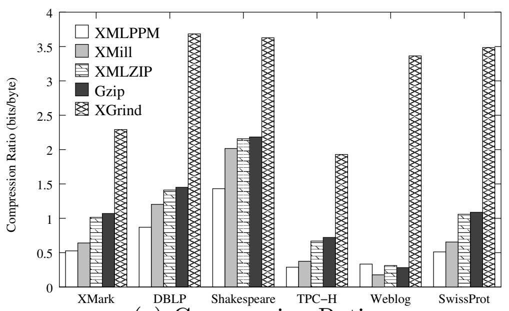
$$\begin{aligned} CR_1 &= \frac{sizeof(compressed\ file) \times 8}{sizeof(original\ file)}\ bits/byte \\ CR_2 &= (1 - \frac{sizeof(compressed\ file)}{sizeof(original\ file)}) \times 100\% \end{aligned}$$

Table 2: Comparing Compression Ratio Expressions CR_1 and CR_2

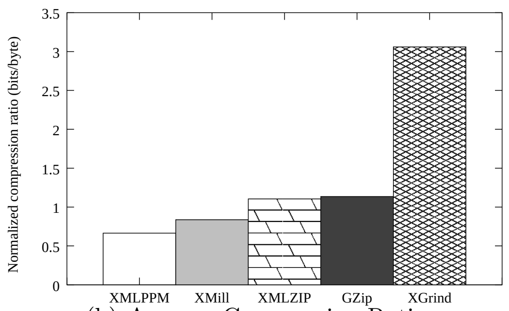
XML Dataset	Doc Size (*KB*)	Compressed Document Size (*KB*)		CR_1 (*bits/byte*)		CR_2 (*percentage %*)	
		Gzip	XMill	Gzip	XMill	Gzip	XMill
Weblog	32722	1156	726	0.282	0.177	96.5	97.8
SwissProt	21254	2889	1739	1.088	0.654	86.4	91.8
DBLP	40902	7418	6149	1.451	1.203	81.9	85.0
TPC-H	32295	2912	1514	0.721	0.375	91.0	95.3
XMark	103636	13856	8313	1.07	0.642	86.6	92.0
Shakespeare	7882	2152	1986	2.184	2.016	72.7	74.8

We now illustrate the difference in both CR definitions by listing the compression ratios achieved by Gzip and XMill in Table 2. CR_2 shows that the fraction of an input document eliminated by Gzip is only a few percent smaller than that of XMill. This means that the performance of Gzip and XMill measured based on CR_2 appear to be similar. However, the actual size of the compressed documents generated by XMill is generally much smaller than that of the Gzip compressed one. For example, the size of the XMill compressed Weblog document is about *60%* of that of Gzip.

This is also true for the SwissProt, TPC-H and XMark documents. On the other hand, as we can see in Table 2, the difference is better reflected by the measurement based on CR_1. For example, we can see from Table 2 that there is an eleven-fold difference between the CR_1 readings for the Weblog (0.177 bits/bytes) and Shakespeare (2.016 bits/bytes) datasets in XMill compression, while the difference between the corresponding CR_2 readings (i.e. 97.8% and 74.8%) is only *23%*. In addition, the notion behind the CR_1 expression (i.e. the number of bits required to represent a byte) gives us intuition related to the *amount of information*, which is a commonly used notion in information theory [40]. Thus, we henceforth choose to adopt CR_1 as the metric to measure compression performance (i.e. the lower CR_1 the better compression).



(a) Compression Ratio



(b) Average Compression Ratio

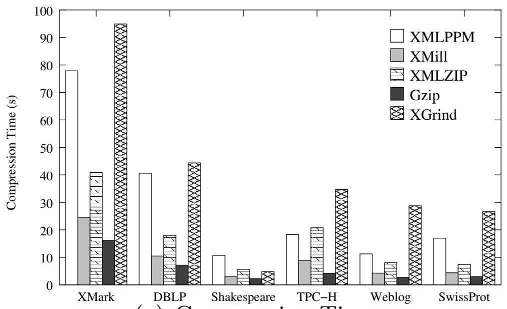
Figure 12: Compression Ratio

Figure 12(a) shows the compression ratio of XMLPPM, XMill, XMLZip, Gzip, and XGrind on the six datasets, respectively. The three unqueriable compressors, XMLPPM, XMill, and XMLZip, achieve compression ratios consistently better than that achieved by the generic text compressor Gzip, while the compression ratio of the queriable compressor XGrind is on average 2.70, 2.77, 3.63, and 4.64 times worse than that of Gzip, XMLZip, XMill, and XMLPPM, respectively. The compression ratios of XMLZip and Gzip are very close to each other for all the datasets, but these two compressors are worse than those of XMill and XMLPPM. XMLPPM achieves the best compression ratio (an average of 0.66 bits per byte) among all the compressors in this experiment. The exceptionally low compression ratio achieved by XMill on Weblog is due to the highly regular structure in Weblog, which makes it fits extremely well in XMill data containers.

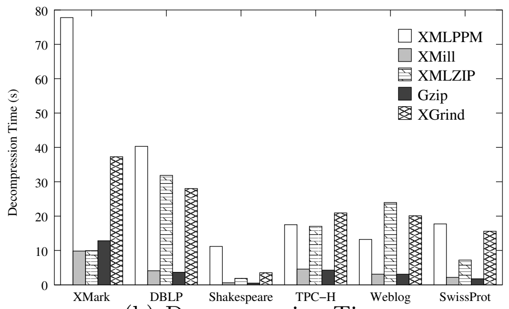
The underlying reason for the good compression ratio achieved by the unqueriable compression technologies is that they utilize the exposed structure information in the input XML document to help restructure the document, which makes it more amenable to compression. On the other hand, in order to support querying over compressed documents, XGrind needs to preserve the XML structure and to compress each XML data item as an individual unit, thus resulting in degradation in the compression ratio. Figure 12(b) helps gain further insight of an overall compression performance for individual compressor. In this chart we simply assume an even weight of the compression ratios with respect to the six datasets given in Figure 12(a) and then compute the average of the compression ratio achieved by the five compressors. Remarkably, other interesting normalization of compression ratio can also be done by assigning unequal weight on the datasets. For example, we may assign different weight in favor to either document-centric or data-centric documents, which is able to reflect the compression performance of a compressor relative to the commonly used data in certain applications.

Among the six compressors, XMLPPM achieves the best compression ratio for all the datasets except Weblog. This could be explained for the following reasons. In XMLPPM, a PPM compression library is adopted to compress the restructured document. Since PPM spends more time exploring and eliminating redundancies of the input document (see the evidence presented in Section 4.2.2 below), it generally achieves a better compression than Gzip. In addition, the underlying PPM compressors in XMLPPM are multiplexed to deal with the input stream effectively (see Section 2.2), which enables XMLPPM to yield a more accurate prediction over the input SAX event tokens and hence boosts its performance in terms of the compression ratio.

4.2.2 Compression and Decompression Times



(a) Compression Time



(b) Decompression Time

Figure 13: Compression and Decompression Times

Figures 13(a) and 13(b) show the compression and decompression time of XMLPPM, XMill, XMLZip, Gzip, and XGrind on the six datasets, respectively. Except for XMill, all the compression technologies require longer compression and decompression times than Gzip. The underlying reasons for this are as follows.

The compression time of XMill is slightly longer than that of Gzip. This is because in its pre-compression phase, XMill separates the structures from the data before applying Gzip to compress the data containers (recall the discussion in Section 2.1). This pre-compression phase introduces a time overhead to the compression process. XMill also needs to reconstruct the original XML structure, with the data items merged into their original position, after decompressing the data containers. Therefore, this extra decompression overhead slightly increases the total decompression time. However, when the XMill-compressed file size is much smaller than the Gzip compressed file size of a given XML size, as shown in the case of the XMark dataset, it is possible that XMill achieves a decompression time that is even shorter than that of Gzip, mainly due to the much smaller disk read overhead.

XMLZip takes more time to compress and decompress an XML document because it uses DOM tree parsing. Both its compression and decompression times are lengthened, since much time is needed to convert the input XML document into a DOM tree in the compression phase, and to convert the DOM tree back to the original document in the decompression phase.

The compression and decompression times of the other two compression technologies, XMLPPM and XGrind, are much slower than those of the other technologies. XMLPPM requires much longer compression and decompression time because it is built on top of PPM, which is intrinsically slower than Gzip. XGrind uses Huffman coding and thus needs an extra parse of the input XML document to collect statistics for a better compression ratio, resulting in almost double the compression time required in a generic compressor, as discussed in Section 3.1.

4.2.3 Memory Usage

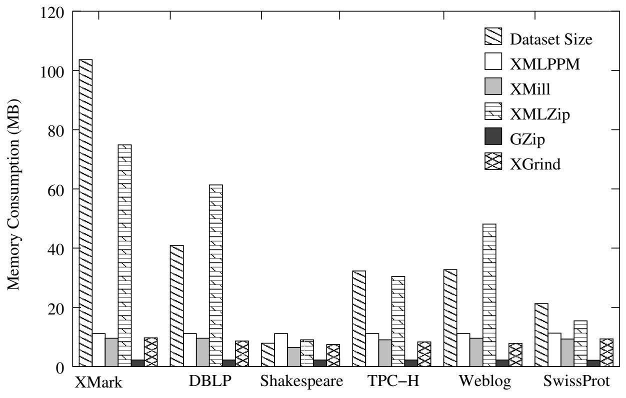


Figure 14: Memory Consumption

Figure 14 shows the maximum size of memory needed for different compressors to run compression on various datasets, where we use the first bar to represent the size of the dataset as a reference. Among all the compressors, Gzip uses the least memory, a fixed 3 MB buffer size, regardless of the input documents. XMill and XGrind use approximately 8 MB of memory over all datasets, while XMLPPM uses about 3 MB more memory than XMill. XMill uses a fixed main memory window size (default 8 MB) and when the memory consumption reaches this fixed size, XMill writes all compressed data to disk and then resumes the compression process. XGrind uses Huffman coding to compress the data and then only needs a small size to maintain the status of the Huffman models. Due to the small overhead of the Huffman models, the total memory consumption does not vary greatly among the datasets. XMLPPM builds four PPM models for all input documents (recall the discussion in Section 2.2); the models have a fixed size limit and therefore the overall memory consumption is also limited. XMLZip parses the XML document using DOM and thus requires the largest memory consumption, which is roughly proportional to the size of the input XML document. This is prohibitive when the input XML document is large.

4.3 Query Coverage and Performance

We analyze the query languages supported by XGrind and XPRESS, and describe them in Extended Backus Naur Form (EBNF), as shown in Figures 15 and 16, respectively. The recent systems XQueC [2] and XQzip [13] claim to be able to support a large part of XQuery [5] and XPath [14] respectively, while XCQ [28] claims to be able to support path queries with multi-predicates and aggregations. Other compression technologies discussed in previous sections (i.e. XMill [30], SCA [29], Millau DDT [42], and XMLPPM [12]) do not support querying, since the generated compressed data is a single unit and therefore must be fully decompressed before any query evaluation.

```
 Q ::= ‘/’ LP OP ‘‘literal’’
LP ::= tag ‘/’ LP | tag (‘/’ @attribute)?
OP ::= > | < | >= | <= | = | !=
```

Figure 15: EBNF for the Query Language Supported by XGrind

```
 Q ::= (‘/’ | ‘//’) LP
LP ::= tag (‘/’ | ‘//’) LP | tag (‘[’ P ‘]’)?
       | tag (‘/’ @attribute)?
 P ::= (tag | @attribute) OP ‘‘literal’’
OP ::= > | < | >= | <= | = | != | range-match
```

Figure 16: EBNF for the Query Language Supported by XPress

From Figures 15 and 16, we can see that XGrind and XPRESS support a very restricted portion of XPath [14]. XGrind only supports path expressions with the *child* and *attribute* axes, while XPRESS only adds the *descendant* axis. The reason for this is that both XGrind and XPRESS adopt homomorphic transformations to preserve the XML structure and their query processing is based on a naive top-down evaluation mechanism. This large preserved structure, together with the naive algorithm, is not efficient enough to support more complex queries.

Admittedly, we are not able to compare the query performance of the various compressors empirically, since at the time of writing, the source code of these compressors, except XGrind, has not been made open. However, we could still obtain some useful numerical data of their query performance evaluation published in the respective papers. According to the results in [46], XGrind's query performance is on average 2 to 3 times better than that of the Native query processor built upon XMill's parser. According to the results in [35], XPRESS's query performance is 2.83 times better than that of XGrind, while XQzip's experimental results in [Cheng and Ng 2004] show that its query performance is 12.84 times better than that of XGrind. XQueC [3] does not compare to any of the compressors for query performance but compared to Galax [33], the results show that XQueC performs better than Galax in many cases.

4.4 An Overall Comparison

We now give an overall comparison of the compression technologies, as listed in Tables 3 and 4, as a summary of our main findings.

With respect to compression performance, XMill has the best average performance, since it achieves a good compression ratio at a comparable speed to that of Gzip and with low constant memory consumption. Although XMLPPM also achieves a very good compression ratio, its compression time is prohibitively longer than that of Gzip, thus hindering its practical applications. XMLZip, SCA, and Millau DDT are generally not preferred to XMill and XMLPPM, since they only achieve a compression ratio slightly better than Gzip and have a much longer compression time. In addition, their memory consumption is proportional to the input data. The other two compressors, XGrind and XPRESS, achieve much worse compression ratio than Gzip; however, they have the benefits of allowing direct evaluation of queries on the compressed data. Both compressors support *exact-match* and *prefix-match* on compressed data, *partial-match* and *range-match* on decompressed data, and the XPath axes *child* and *attribute*; while XPRESS also supports *range-match* on compressed numeric data and the XPath axis *descendant*.

5 Desirable Features of XML Compression

In this section, we propose some desirable quality features for XML conscious compression technology.

1. *Effective Compression.*
   Effective XML compression requires a good performance in compression ratio, compression and decompression times, and memory consumption. An effective XML compressor should be able to achieve consistently, over various types of XML data, a compression ratio better than (for the case of unqueriable compression) or comparable to (for the case of queriable compression) that of the generic text compressor Gzip; while it should also achieve compression and decompression times comparable to those of XMill.

   According to information theory [40], two information sources carrying the same messages should have the same degree of entropy; thus an equal ultimate compressed size can be obtained. In practice, a good compression ratio is achievable only if there is adequate exploring and exploiting of the redundancy in the XML structure and the commonality in the XML data. Good

Technologies	Compression Ratio (compared with Gzip)	Compression Time (compared with Gzip)	Memory Usage (for compression)	Underlying Compression Scheme Used	Query Coverage	Parser Used
XMill	Consistently Better	Slightly Slower	Constant 8 MB (default)	Gzip (default), Bzip, Compress (UNIX)	Not Support Querying	SAX
XGrind	Much Worse	At least two times longer	Roughly Constant	Huffman Coding	Exact-match, Prefix-match, Xpath Axes: Child Attribute	SAX
XPRESS	Much Worse	At least two times longer	Roughly Constant	Huffman Coding, Approximated Reverse Arithmetic Encoding	Exact-match, Prefix-match, Xpath Axes: Child and Descendant Attribute	SAX
XMLPPM	Much Better	Prohibitively Longer	Constant	PPM	Not Support Querying	SAX
XMLZip	Comparable	Much Longer	Proportional To Input Data Size	Gzip	Not Support Querying	DOM
SCA	Slightly Better	Much Longer	Proportional To Input Data Size	Structure Compression Algorithm	Not Support Querying	DOM
Millau DDT	Slightly Better	Much Longer	Proportional To Input Data Size	Differential DTD Tree Compression, Gzip	Not Support Querying	DOM

Table 3: Comparison of XML Conscious Compression Technologies (Part I)

Technologies	Platform	Evaluation Method	Code Availability
XMill	Unix/Windows (Implemented in C++)	Empirical	Yes
XGrind	Unix/Windows (Implemented in C++)	Empirical	Yes But Not Fully Portable
XPRESS	Portable (Implemented in Java)	Reference	No
XMLPPM	Unix/Windows	Empirical	Yes
XMLZip	Portable (Implemented in Java)	Empirical	Yes (Binary Code Only)
SCA	—	Reference	No
Millau DDT	Portable (Implemented in Java)	Empirical	No

Table 4: Comparison of XML Conscious Compression Technologies (Part II)

compression and decompression times are achievable only by parsing and processing the XML document just once, such as by using an SAX parser.

The memory consumption should also be taken into consideration of effective compression, since an XML document can be huge and it is infeasible to load the entire document into the memory to perform compression or decompression. Therefore, there should be a limit to the main memory window size for the compression and decompression processes. In addition, the window size should be *independent* of the size of the input document and preferably be a constant. This also implies that parsing the input document using a DOM parser is prohibitive and an SAX parser should always be used instead.

2. *Expressive Query Language and Efficient Querying Engine.*
   The recently proposed XML compressor is capable of evaluating a range of common XML queries over the compressed XML data. Ideally, the compressor should support an expressive query language as well as an efficient query evaluation. XPath [14] and XQuery [5] are the two most commonly supported querying languages and they are also the standard XML query languages proposed by the W3C. Therefore, a compressor should be able to support one of these two well-established languages or a large fragment of them. The challenge of supporting XQuery over compressed data is that it requires semantics for connecting nodes on paths, such as a natural join for the *For clause*, a semi-join for the *Where clause*, and an outer-join for the *Return clause* in the FLWR expressions [5].

   A main contributor to the querying cost for compressed data is the decompression overhead. A general approach, as discussed in Sections 3.1 and 3.3, is to compress each data item individually, such as those proposed in [46, 35]. Decompression can then be avoided by encoding string conditions to match the compressed data items. However, the Huffman algorithm [25], as used in [46, 35], supports only equality predicates, while decompression is still inevitable for any other inequality predicates. To support inequality predicates without decompression, the ALM compression algorithm [1], as adopted in XQueC [2], can be used. However, in general, compressing data items individually degrades compression ratio and increases compression time. Another approach proposed recently [28, 13] is to compress the XML data into small blocks and then manage a buffer pool for the decompressed blocks. This avoids the same block being repeatedly decompressed during the query evaluation, as discussed in Section 3.5. However, it is difficult to choose an optimal block size due to the different characteristics of XML documents and queries.

3. *Minimal User Intervention and Auxiliary Structures*
   A good compression technology should be general enough to perform compression; that is, a minimal level of special aids, such as DTD and XML schema, is needed. Among existing XML compression techniques, XMill is the only one that uses specialized semantic compression algorithms. However, in XMill the container expressions are specified manually by user and thus it hinders the application of semantic compressions to obtain a better compression ratio. As XML schema has become more stable standard after several years of development by the World Wide Web Consortium, it can be employed to extract the data-type information and the paths of the data in order to automatically select specialized semantic compression algorithms.

   To support efficient query processing and an expressive query language, an efficient auxiliary structure, such as the SIT structure proposed in XQzip [13] or the structure summary used in XQueC [3], can be employed to aid the efficient access to a required data item. However, the extra data structure adds to the total output file size and total processing time, which degrade the overall compression ratio and processing time, respectively. Therefore, such an auxiliary structure should be devised strategically.

It should be noted that the features listed above may conflict with each other. For example, compressing XML data into blocks [30] generally achieves a good compression ratio but full decompression is needed to query the compressed data. On the other hand, compressing the data items individually [46, 35, 2] can avoid full decompression in query processing, but usually degrades the compression ratio and time. Auxiliary data structures can be used to support efficient query evaluation and an expressive query language; however, they increase the total file size and require an extra amount of processing time to construct and load into memory for query evaluation.

We agree with the view in [12] that the compression ratio and the compression time are two conflicting factors, and that a better compression ratio can generally be achieved if the user is willing to accept a longer compression time. For example, in XMill and XCQ, a compression ratio that is of 10% to 30% better can also be achieved by using Bzip [44] instead of Gzip as the underlying compression library. However, Bzip is much slower than Gzip (several times in magnitude) when used as the underlying compression library in a compressor. A similar trade-off between compression ratio and time is also found in [53].

Despite all these challenges, we believe that for more advanced XML compression-and-querying systems, innovative technologies are able to reach a better compromise, getting closer to the desirable quality features of effective compression, an expressive query language and an efficient querying engine, and minimal auxiliary structure and user intervention.

6 Deploying Compression Techniques

From the point of view of applicability, compression techniques should not be an isolated component in a real-world information system. We now discuss some deployment issues related to the compression technology.

Suppose in a proxy server architecture [36], an XML request is sent from a client via a browser. The request is compressed by the client-side proxy and then sent to the server. The sever-side proxy decompresses the request before it is sent to the server. The response from the server can be returned using the similar processes in the architecture. In this setting the cost-effectiveness depends on the compression/decompression overhead, the average size of the documents and the effectiveness of a programmable proxy server packages for handling arbitrary XML document compression.

In addition to applying the compression techniques in the scenario of data exchange over the Web, another important area of XML compression is on a native XML DBMS. In fact, compression techniques have been studied extensively in the context of relational data [32, 51]. By compressing XML documents in the database, we are able to obtain a significant advantage in reducing the I/O cost. In this case, the reduction of I/O is often more significant than the overhead of decompression, since the disparity of CPU and I/O is increasing fast. For example, Tamino 2.3 XML databases [45] gain performance benefits from supporting field compression inside the database system, which uses some proprietary compression algorithms such as Java built-in "JAR" or generic Gzip.

The choice of the compression techniques also depends on the application in which the technologies are to be embedded. For example, in applications such as data archiving, where querying is infrequent, and data exchange, when data are filed as LOBs or large documents, XMill is desirable for the reason that it is able to reduce greater space requirement and better utilize the network bandwidth. It is demonstrated in [8] that, when XMill is applied to archive scientific data such as SwissProt, the compressor can obtain a better compression ratio than general compression tools like gzip. It is also worth mentioning that when compression and decompression speeds are not a concern, XMill's compression ratio can be further improved by adopting Bzip [44] as its underlying compression library.

On the other hand, if querying is frequent and the XML data are managed in a fine-granular way such as in an underlying XDBMS, XQueC can be embedded in such applications, since XQueC allows efficient access to individual compressed values and outperforms its counterpart fine-grained compression techniques, XGrind and XPRESS, in both query expressiveness and query performance. Moreover, XQueC addresses the embedment of compression in XML databases and is thus the most feasible choice among existing compression techniques for applications that operate on a backend database system.

If we strike for a balance between the features of XMill and XQueC in some systems, XQzip is a good choice for those applications that do not heavily rely on or cannot afford using a secondary storage system. In the meantime the applications may still require fast response to (path) queries and a good compression ratio. For example, in order to exchange information in a peer-to-peer network of Pocket-PCs, for which a secondary storage is obviously not feasible, we can apply XQzip to compress the XML data into a set of small blocks which can be transferred and queried among the peers.

7 Concluding Remarks

We recognize that the *size problem* already hinders the adoption of XML, since in practice, it substantially increases the costs of data processing, data storage, and data exchanges over the Web. We believe that XML-conscious compression is one of the key solutions to the problem. With limited resources, it has not been our intention to cover all the activities concerning XML data compression in this article. Our purpose is to stimulate interest among the involved communities by presenting an overview of the development.

We surveyed and analyzed the efforts that have been made towards developing better strategies and technologies for compressing XML data. We classified the following eight XML-conscious compression technologies: (1) *XMill*, (2) *XMLPPM*, (3) *SCA*, (4) *Millau*, (5) *XMLZip*, (6) *XGrind*, (7) *XPress*, and (8) *XML Skeleton Compression* [7], into two categories of queriable and unqueriable XML compression technologies. We also briefly discussed three recently proposed query-aware XML compressors, XQueC, XQzip and XCQ, in Section 3.5. We discussed the working principles and highlighted the features of the queriable and unqueribale XML compression in Sections 2 and 3, respectively. We made a detailed comparison in Section 4 of XMill, XGrind, XMLPPM, and XMLZip, with reference to a common generic compressor GZip. We identified a set of desirable features for XML-conscious compression and discussed the challenges in Section 5.

Research on XML compression is related to a wide range of computer science disciplines such as databases, information retrieval, information theory, and algorithm studies. We feel that there are still a number of issues to be addressed by both academic researchers and practitioners. The following issues in particular are interesting and challenging (though by no means exhaustive).

1. It is known that from a theoretical point of view, XML data compression, like usual data compression, is determined by the degree of entropy of the data. However, in addition to using the syntactic aspect of data, we could explore the use of semantics or workload statistics in compression. How can data semantics in XML and query workload statistics be exploited in order to make an XML compressor adaptive and more effective to an application domain?

2. From the existing research work, we know that different kinds of trade-offs exist among the different features of compressors. The most apparent one is the trade-off between the compression ratio and the compression time. Another one is the trade-off between the compression ratio and the decompression overhead incurred in query processing of the full-chunk compression and the fine-grain compression. If auxiliary data structures are used to support efficient querying, the trade-off between the total file size and the querying performance is another concern. Therefore, a challenge will be to address the possible trade-offs between the compression quality (lossless, compression ratio, etc) and speed, and the query performance and query expressiveness.

3. The queriable compressors are themselves strengthened to support efficient querying over compressed XML data. A natural step to advance the compressor is to address the following issue. How can the updating operations over compressed data be carried in an effective manner?

4. It seems clear from the work in XQueC and XQzip that auxiliary structures are desirable in querying compressed XML data. However, we think that there are still rooms for further improvement in terms of reducing the resource overheads. How can a more effective auxiliary structure, such as an indexing scheme, be devised to aid querying compressed XML data?

5. A comprehensive cost model that takes into account various factors, such as query selectivity, the data chunk size of compressed data, and document characteristics, will be useful when running queries over compressed data. How can an analytical model for querying compressed data, which should help to optimize the query engine of a compressor be established?

To this end, almost all the compression technologies are studied as "standalone" systems. Only Millau explicitly presented two *e*-commerce applications that adopt the compressor in [42]. As a result, the usability of compression technologies are not clear in the context of modern Web applications. In fact, different compressed document formats are used in different compression technologies. So it is useful to study the issues concerning how users are able to adopt the developed compressors in different services and how the compressors are interoperable in a Web architecture, which may consist of Web clients and servers, mediators, and Pocket PCs.

Acknowledgement: The authors would like to thank the anonymous referees for their insightful and encouraging comments.

References

[1] G. Antoshenkov. Dictionary-Based Order-Preserving String Compression. *VLDB Journal 6*, page 26-39, (1997).

[2] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. Efficient Query Evaluation over Compressed XML Data. *Proceedings of EDBT* (2004).

[3] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. XQueC: Pushing Queries to Compressed XML Data. *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, (2003).

[4] T. Bell, J. Cleary, and I. Witten. Text Compression. *Prentice Hall, Englewood Cliffs, New Jersey*, (1990).

[5] S. Boag et al. XQuery 1.0: An XML Query Language, Nov. (2002). `http://www.w3.org/TR/xquery`.

[6] J. Bosak. Shakespeare 2.00.
`http://www.cs.wisc.edu/niagara/data/shakes/shakspre.htm`.

[7] P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML. *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, May (2003).

[8] P. Buneman, S. Khannay, K. Tajimaz, W. C. Tan. Archiving Scientific Data. In *Proceedings of SIGMOD* (2002).

[9] M. Burrows and D.J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. *Technical Report, Digital Equipment Corporation, Palo Alto, California* (1994).

[10] M. Cannataro, C. Comito, and A. Pugliese. Squeeze X: Synthesis and Compression of XML Data. *IEEE Proceedings of the International Conference on Information Technology: Coding and Computing* (2002).

[11] M. Cannataro, C. Gianluca, A. Pugliese and D. Sacca. Semantic Lossy Compression of XML Data. *The 8th International Workshop on Knowledge Representation Meets Databases* (2001).

[12] J. Cheney. Compressing XML with Multiplexed Hierarchical PPM Models. *Proceedings of the IEEE Data Compression Conference*, pp. 163-172 (2000).

[13] J. Cheng and W. Ng. XQzip: Querying Compressed XML Using Structural Indexing. *Proceedings of EDBT* (2004).

[14] J. Clark. XML Path Language (XPath), (1999).
`http://www.w3.org/TR/xpath.`

[15] J. Cleary, W. Teahan, and I. Witten. Unbounded Length Contexts for PPM. *Proceeding of the IEEE Data Compression Conference*, pp. 52-61, March (1995).

[16] J. G. Clearly and I. H. Witten. Data Compression Using Contexts for PPM. Computer Journal, Vol. 40, Nos (2/3): pp. 67-75 (1997).

[17] DBLP. `http://dblp.uni-trier.de/`.

[18] Document Object Model (DOM) Level 2 Specification Version 1.0, W3C Recommendation, November (2000). `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113`.

[19] Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation, October (2000). `http://www.w3.org/TR/REC-xml/`.

[20] J. Gailly and M. Adler. gzip 1.2.4. `http://www.gzip.org/`.

[21] M. Girardot and N. Sundaresan. Efficient Representation and Streaming of XML Content over the Internet Medium. *IEEE International Conference on Multimedia and Expo (I)*, pp. 67-70 (2000).

[22] M. Girardot and N. Sundaresan. Millau: An Encoding Format for Efficient Representation and Exchange of XML over the Web. *Proceedings of the 9th International WWW Conference*, pp. 747-765, May (2000).

[23] R. Goldman and J. Widom. DataGuide: Enabling Query Formation and Optimization in Semistructure Databases. *Proceedings of the International Conference on Very Large Data Bases*, pp. 436 - 445, Athens, Greece, August (1997).

[24] H. Hopcroft and J. Ullman. Introduction to Automata Theory, langauges, and Computation. *Addison-Wesley*, (1979).

[25] D. A. Huffman. A Method for Construction of Minimum-Redundancy Codes. *Proceeding of the IRE*, (1952).

[26] H. Ishikawa, S. Yokoyama, S. Isshiki, and M. Ohta. Project Xanadu: XML- and Active-Database-Unified Approach to Distributed E-Commerce. *Proceeding of the 12th International Workshop on Database and Expert Systems Applications*, September (2001).

[27] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexes for Branching Path Queries. *Proceedings of SIGMOD* (2002).

[28] W. Y. Lam, W. Ng, P. T. Wood, and M. Levene. XCQ: XML Compression and Querying System. *Poster Proceedings, 12th International World-Wide Web Conference (WWW2003)*, May (2003).

[29] M. Levene and P. T. Wood. XML Structure Compression. *Proceedings of the Second International Workshop on Web Dynamics*, May (2002).

[30] H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 153-164 (2000).

[31] Log Files - Apache HTTP Server. `http://httpd.apache.org/docs/logs.html`.

[32] W. Ng and C. Ravishankar. *Block-Oriented Compression Techniques for Large Statistical Databases* IEEE TKDE 9(2): 314-328 (1997).

[33] A. Marian and J. Simeon. Projecting XML Documents. *Proceedings of VLDB* (2003).

[34] J. M. Martinez. MPEG-7 Overview (version 9). `http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm`.

[35] J. K. Min, M. J. Park, and C. W. Chung. XPRESS: A Queriable Compression for XML Data. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2003).

[36] W. Ng. Evaluating the Client Side Approach and the Server Side Approach to the WWW and DBMSs Integration. *Proceedings of the 9th International Database Workshop, Heterogeneous and Internet Databases IDW'99*, pp. 72-82 (1999).

[37] pkzip. `http://www.pkware.com/`.

[38] SAX. `http://www.saxproject.org/`.

[39] A. R. Schmidt and F. Waas and M. L. Kersten and M. J. Carey and I. Manolescu and R. Busse. XMark: A Benchmark for XML Data Management. *Proceedings of VLDB*, (2002).

[40] T. M. Cover and J. A. Thomas. Elements of Information Theory. *WILEY-INTERSCIENCE, John Wiley & ons, Inc., New York*, (1991).

[41] A. Silberschatz and P. Galvin. Operating Systems Concepts. *Addison-Wesley, 5th Edition* (1998).

[42] N. Sundaresan and R. Moussa. Algorithms and Programming Models for Efficient Representation of XML for Internet Applications. *Proceedings of the 10th International WWW Conference*, pp. 366-375, May (2001).

[43] SWISS-PROT Protein Knowledgebase. `http://www.expasy.ch/sprot/`.

[44] The bzip2 and libbzip2 official home page. `http://sources.redhat.com/bzip2/`.

[45] Software AG: Tamino XML Databases.
http://www.softwareag.com/tamino.

[46] P. M. Tolani and J. R. Haritsa. XGRIND: A Query-friendly XML Compressor. *IEEE Proceedings of the 18th International Conference on Data Engineering* (2002).

[47] TPC-H: An ad-hoc, decision support benchmark.
http://www.tpc.org/tpch/default.asp.

[48] J.L. Tzeng. Transferring Data between XML Documents and PostgreSQL DBMS. http://www.cs.indiana.edu/ jetzeng/jt-xmldb/.

[49] WAP Binary XML Content Format, W3C NOTE, June (1999).
http://www.w3c.org/TR/wbxml/.

[50] M. A. Weiss. Data Structure and Algorithm Analysis in C++. *Addison-Wesley*, 2nd Edition (1999).

[51] T. Westmann, D. Kossmann, S. Helmer and G. Moerkotte. *The Implementation and Performance of Compressed Databases.* SIGMOD Record 29(3): 55-67, (2000).

[52] Winzip. http://www.winzip.com/.

[53] XMLZip - XML Solutions. http://www.xmls.com/.

[54] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, Vol. IT-23, No. 3, pp. 337-343, May (1977).