XGRIND: A Query-friendly XML Compressor

Pankaj M. Tolani Jayant R. Haritsa*
Dept. of Computer Science & Automation
Indian Institute of Science, Bangalore 560012, INDIA

Abstract

XML documents are extremely verbose since the "schema" is repeated for every "record" in the document. While a variety of compressors are available to address this problem, they are not designed to support direct querying of the compressed document, a useful feature from a database perspective. In this paper, we propose a new compression tool called `XGrind`*, that directly supports queries in the compressed domain. A special feature of* `XGrind` *is that the compressed document retains the structure of the original document, permitting reuse of the standard XML techniques for processing the compressed document. Performance evaluation over a variety of XML documents and user queries indicates that* `XGrind` *simultaneously delivers improved query processing times and reasonable compression ratios.*

1. Introduction

The XML language [1], by virtue of its self-describing and textual nature, has become extremely popular as a medium of data exchange and storage, especially on the Internet. To support this functionality, XML resorts to, in database terms, storing the "schema" with each and every "record" in the document. This is in marked contrast to the traditional database approach of storing the meta-data once for the whole database. A consequence of XML's repeating-schema characteristic is that documents are extremely verbose as compared to their intrinsic information content. In fact, according to a recent industry white-paper [19], the typical size increase is estimated to be as much as 400 percent!

One approach to address the verbosity problem is to utilize a standard text compressor, for example, `gzip` [17], and thereby reduce the size of the document. An alternative is to design an XML-specific compressor – this approach resulted in the `XMill` tool, proposed recently by Liefke and Suciu [10]. `XMill` achieves compression ratios typically in excess of 80 percent on large XML documents by grouping semantically related data items into "containers", separately compressing each container with a specialized compressor ideal for that container, followed by a `gzip` on each container. For example, the meta-data (in the form of XML tags and attributes) and the data (element/attribute values) are compressed separately. A performance study [10] showed `XMill` to consistently provide better compression ratios than `gzip`.

Since `XMill` is designed to minimize the *size* of the compressed XML document, it is attractive in terms of reducing the network bandwidth required for transmission, and the disk space required for storage, of the original document. However, its compression approach is not intended for directly supporting *querying* or *updating* of the compressed document. In fact, accomplishing such operations on XMill-compressed documents would typically entail a *complete decompression* of the file.[1]

The ability to perform direct querying is important for a variety of applications, especially for those hosted on resource-limited computing devices such as Palm Tops. For example, consider a vendor who travels around with a detailed list of her customers and orders, in compressed XML format, on her PDA. She could be reasonably expected to frequently query this database in order to check customer contact information, order status, delivery schedules, etc., as well as enter information about new customers or orders, status updates, etc. If she would need to decompress the entire document every time she wanted an answer or needed to make an update, it could be quite time-consuming and tiresome. Worse, it may even turn out to be *impossible* to perform the decompression since her device may run out of space to hold the uncompressed document!

At the other extreme of the resource spectrum, data warehouses storing XML documents may find that, even if decompressing were available *for free*, directly supporting

*Contact Author: haritsa@dsl.serc.iisc.ernet.in

[1] Since XMill compresses in "chunks" of 8MB size, in principle it is possible to separately decompress and query each chunk – however, there are significant design and implementation complexities involved in this process, as mentioned in [11].

IEEE COMPUTER SOCIETY

data-intensive decision support queries on the compressed data may result in a significant improvement in query response times as compared to querying the uncompressed version. This is because compression, as highlighted in [5, 6, 9, 13], provides many other benefits apart from the obvious utility of reduced space: disk seek times are reduced since the compressed data fits into a smaller physical disk area; disk bandwidth is effectively increased due to the increased information density of the transferred data; and, the memory buffer hit ratio increases since a larger fraction of the document now fits in the buffer pool.

1.1. The XGRIND Compressor

Based on the above observations, we propose in this paper a new XML compression tool, called **XGrind**, that directly supports queries in the compressed domain. It compresses at the granularity of individual element/attribute values using a simple context-free compression scheme based on Huffman coding [8]. This means that *exact-match* and *prefix-match* user queries can be *entirely* executed directly on the compressed document, with decompression restricted to only the final results provided to the user.[2] Further, *range* or *partial-match* queries require on-the-fly decompression of only those element/attribute values that feature in the query predicates, not the entire document.

A novel and especially useful feature of `XGrind` is that it *retains the structure of the original XML document* in the compressed format also. This means that the compressed document can be parsed using exactly the same techniques that are used for parsing the original XML document. A related major benefit is that *XML indexes* [12] can be created on the compressed document. Further, *updates* to the XML document can be directly executed on the compressed version. Lastly, a compressed document can be checked for *validity* against the compressed version of its DTD. We expect that these properties would be of considerable utility in practical settings, especially those hosting large numbers of XML documents. For example, major repositories of genomic data such as the European Bioinformatics Institute (EBI) [16], allow registered users to upload new genetic information to their archives. It would be extremely useful if such information could be compressed by the user and then uploaded, checked for validity, and integrated with the existing archives, all operations taking place completely in the compressed domain.

Another feature of `XGrind` is that, for XML documents adhering to a DTD, it attempts to utilize the information in the DTD to enhance the compression ratio. For example, attribute values that are of enumerated-type are recognized from the DTD and are encoded differently from other attribute values.

[2]Note that this decompression is the minimum which will have to be performed by any compression scheme.

1.2. Performance Results

We have conducted a detailed performance evaluation of `XGrind` over a representative set of real and synthetic XML documents, including some generated from Xmark [24], the recently announced XML benchmark, for a variety of XML search queries. Our study considers a variety of metrics including the compression ratio, the compression time, and the query processing times. To our knowledge, there do not exist any prior queryable XML compressors. Therefore, we have attempted to place the `XGrind` performance results in perspective as follows: (a) For the compression ratio and compression time metrics, we compare with the `XMill` compressor; (b) For the query processing time metric, we compare against a query processor, hereafter referred to as `Native`, which is built around XMill's XML parser and operates directly on the original uncompressed document.

Our experimental results show that `XGrind` simultaneously and efficiently achieves a reasonably good compression ratio compared to `XMill` and substantially improved query processing times with regard to `Native`.

2. Background Material

In this section, we overview background material on text compression techniques, and on the `XMill` compressor, which represents the state-of-the-art in XML compression.

Most lossless[3] data compression techniques are based on one of two models: **statistical** or **pattern**.[4] With statistical modeling, each distinct *character* of the input data is encoded, with the code assignment based on the probability of the character's appearance in the data. In contrast, pattern-based compression schemes recognize duplicate *strings* in the input data, and these duplicates are replaced either by pointers to the first appearance of the string, or by an index into a dictionary that maps strings to codes.

Another dimension of lossless compression algorithms is that they may be **adaptive** or **non-adaptive**. In adaptive schemes no prior knowledge about the input data is assumed and statistics are dynamically gathered and updated during the encoding phase itself. On the other hand, non-adaptive schemes are essentially "two-pass" over the input data: during the first pass, statistics are gathered, and in the second pass, these values are used for encoding.

Most of the popular compression tools are based on one of the following algorithms: **Huffman**, **Arithmetic**, **LZ77**

[3]Only lossless techniques are considered viable for XML compression since the documents contain textual information.

[4]An exception is the classical run-length encoding scheme.

IEEE COMPUTER SOCIETY

or **LZ78**. The Huffman and Arithmetic coding techniques implement the statistical model, while LZ77 and LZ78 are pattern-based. For Huffman and Arithmetic, both adaptive and non-adaptive flavors are available, whereas both the LZ encoders are adaptive. The compressors evaluated in this paper utilize the Huffman and LZ77 techniques, whose details are described next.

2.1. Huffman Coding

In Huffman coding [8], the most frequent characters in the input data are assigned shorter codes and the less frequent characters are assigned longer codes. The longer codes are constructed such that the shorter codes do not appear as prefixes. In particular, a tree is constructed with the characters of the input alphabet forming the leaves of the tree. The links in the tree are labeled with either `0` or `1` and the code for a character is the label sequence that is obtained by traversing, in the Huffman tree, the path from the root to the leaf node corresponding to that character.

In non-adaptive Huffman coding, the Huffman tree is completely built before encoding starts, and remains unchanged during the encoding process. Adaptive Huffman coding, on the other hand, starts off with a Huffman tree that is built using an *assumed* frequency distribution of the characters in the data. As the encoding process proceeds and more data is scanned, the Huffman tree is modified based on the data seen up to that point. Thus the same character can have different codes depending on its location in the data being compressed (unlike non-adaptive Huffman).

2.2. LZ77 Coding

The LZ77 coding scheme [18] is used in popular compression tools such as `gzip`. Here, the input data is scanned sequentially and the longest *recognized* input string (that is, a string which already exists in the string table) is parsed off each time. The recognized string is then replaced by its associated code. Each parsed input string, when extended by its next input character, gives a string that is not yet present in the string table. This new string is added to the string table and is assigned a unique code value. In this manner, the string table is built incrementally during the compression process. For decompression, the decoder logically uses the same string table as the encoder and constructs it incrementally in a similar manner.

2.3. The XMill Compressor

The `XMill` [10] compressor, as mentioned earlier, represents the state-of-the-art in XML compression. In XMill's document model, each XML document is composed of three kinds of tokens: *tags*, *attributes*, and *data values*. These tokens are organized as a tree, with internal nodes being labeled with tags or attributes, and leaves labeled with data values. The path to a data value is the sequence of tags, (and, possibly one attribute) from the root to the data value node.

With the above model, `XMill` operates in the following manner: First, meta-data in the form of XML tags and attributes is compressed separately from the data, which is the set of strings formed from element and attribute values. Second, semantically related data items are grouped into "containers". For example, all `<name>` data items form one container, while all `<phone>` items form a second container. This is an extension to the semi-structured domain of the notion of column-wise or domain-wise compression that is well-known in relational DBMS (e.g. [9, 13]). The motivation for such semantic grouping is that data belonging to the same group will usually have similar characteristics and can therefore be compressed better than data sequences that have only syntactic proximity. Third, each container is compressed separately with a specialized compressor that is ideal for that container. For example, a delta (difference) compressor may be used for a container hosting integers that typically have moderate changes from one value to the next, while a run-length encoder may be used for domains with a very limited set of values (e.g., "Male" or "Female" for a gender element). Finally, the outputs of all containers are individually compressed using `gzip`, which as mentioned above, is based on LZ77, and the results are concatenated into a single XML file.

A performance study over a wide variety of XML documents showed `XMill` to consistently provide improved compression ratios as compared to plain `gzip`, which treats the entire file as a continuous stream of bytes and does not associate any semantics with the contents.

3. The XGRIND Query-friendly Compressor

In this section, we first describe the features of `XGrind`, our new XML compressor. These features are intended to ensure both good query performance and reasonable compression ratios. We conclude with a presentation of `XGrind`'s architectural and implementation details.

3.1. Compression Techniques

`XGrind` uses different techniques for compressing meta-data, enumerated-type attribute values, and (general) element/attribute values, respectively. These techniques are described below:

IEEE COMPUTER SOCIETY

3.1.1 Meta-Data Compression

XGrind follows the XMill compression approach of separating structure from content. The method to encode meta-data is similar to that in XMill, and is as follows: Each start-tag of an element is encoded by a 'T' followed by a uniquely assigned element-ID. All end-tags are encoded by '/'s. Attribute names are similarly encoded by the character 'A' followed by a uniquely assigned attribute-ID.

3.1.2 Enumerated-type Attribute Value Compression

Enumerated-type attribute values are a common occurrence in XML documents. For example, the states of a country, or the set of departments in a company, or the set of zip-codes, are all instances of frequently occurring enumerated-type attribute values. This knowledge is often captured in the DTD itself. XGrind identifies such enumerated-type attributes by examining the DTD of the document and encodes their values using a simple $log_2 K$ encoding scheme to represent an enumerated domain of K values.

3.1.3 General Element/Attribute Value Compression

While the above schemes cater to meta-data and enumerated-type attribute values, we now move on to the compression technique for general element/attribute values, which typically form the bulk of the XML document.

Given XGrind's goal of efficiently querying compressed XML documents, a *context-free* compression scheme is required. That is, a compression scheme in which the code assigned to a string in the document is independent of its location in the document. This feature allows us, given an arbitrary string, to locate occurrences of that string in the compressed document directly, without decompressing it. This is done by first compressing the query string (expressed as a path expression) and then searching for occurrences of its corresponding encoded sequence in the compressed document.

Context-free compression is *not* possible with adaptive algorithms such as LZ77, since the code assigned to a data item is dependent on the entire contents of the document prior to the occurrence of the data item. That is, only with a complete decompression of the prior contents is it possible to match a sequence. On the other hand, context-free coding of strings is possible with the *non-adaptive* versions of compression algorithms such as Huffman coding and Arithmetic coding. We have currently implemented the non-adaptive Huffman compression algorithm in XGrind. To support the non-adaptive feature, two passes have to be made over the XML document: the first to collect the statistics and the second to do the actual encoding.

In principle, we could use a single character-frequency distribution for the entire document. However, in XGrind, we compute a *separate* frequency distribution table for *each* element and non-enumerated attribute. The motivation for this approach is that data belonging to the same element/attribute is usually semantically related and is expected to have similar distribution. For example, data such as telephone numbers or zip-codes will be composed exclusively of digits. Therefore, the characteristics of each element/attribute are reflected more accurately and the smoothing out of the peculiarities of a particular element/attribute (which may happen in the case of a single document-wide frequency distribution) is prevented.[5] Since we expect that queries will typically have predicates related to element/attribute values, we compress at the granularity of individual element/attribute values. This is done during the second pass using the set of frequency tables generated during the first pass.

With the above scheme, queries can be carried out over the compressed document without fully decompressing it. More precisely, *exact-match* (the search key is a specific data value) and *prefix-match* (the search key is a prefix of the data values) queries can be *completely* carried out directly on the compressed document, while *range* (the search key covers a range of data values) or *partial-match* (the search key is a substring of the data values) queries require *on-the-fly* decompression of only the element/attribute values that are part of the query predicates.

3.2. Homomorphic Compression

The most novel feature of the XGrind compressor is that its output, like its input, is *semi-structured* in nature. In fact, the compressed XML document can be viewed as the original XML document with its tags and element/attribute values replaced by their corresponding encodings. The advantage of doing so is that the variety of efficient techniques available for parsing/querying XML documents can also be used to process the *compressed document*. Second, *indexes*, such as those proposed in [12], can now be built on the compressed document in similar manner to those built on regular XML documents. Third, *updates* to the XML document can be directly executed on the compressed version. Finally, a compressed document can be checked for *validity* against the compressed version of its DTD, without having to resort to any decompression, as shown by the following property.

Given an XML document $\mathcal{X}$ which is valid for a DTD $\mathcal{D}$, let $h_{\mathcal{D}}$ be the homomorphism defining the XGrind encoding scheme for the meta-data and enumerated-type attribute values. Let $h_{\mathcal{D}}(\mathcal{D})$ denote the compressed DTD and $h_{\mathcal{D}}(\mathcal{X})$ denote the compressed XML document. The follow-

[5] This is similar to collecting column or domain statistics for compression in an RDBMS [13].

IEEE COMPUTER SOCIETY

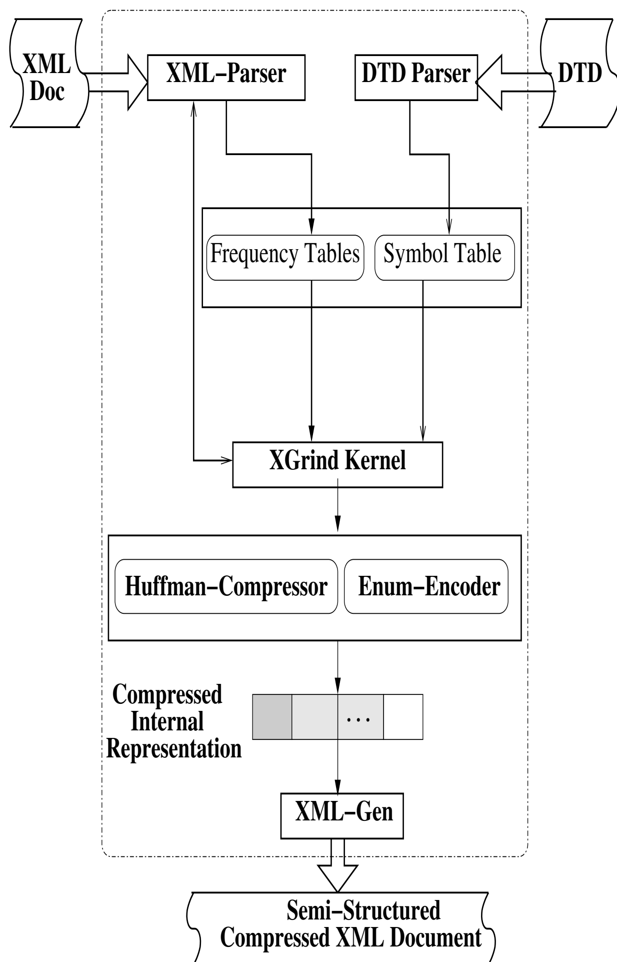

Figure 1. Architecture of `XGrind` Compressor

ing property is a consequence of the "context freeness" of the compression scheme and the semi-structured nature of the output.

$$\mathcal{X} \textit{ is valid for } \mathcal{D} \Leftrightarrow h_{\mathcal{D}}(\mathcal{X}) \textit{ is valid for } h_{\mathcal{D}}(\mathcal{D}).$$

In other words, the `XGrind` compressed document is valid with respect to its associated compressed DTD. The proof for this follows from the closure of regular languages and context-free languages under homomorphisms and inverse homomorphisms [7].

3.3. System Architecture

The architecture of the `XGrind` compressor, along with the information flows, is shown in Figure 1. The **XGrind Kernel** is the heart of the compressor. It starts off by invoking the **DTD Parser**, which parses the DTD of the XML document, initializes *frequency tables* for each element or non-enumerated attribute, and populates a *symbol table* for attributes having enumerated-type values. The kernel then invokes the **XML Parser**, which scans the XML document and populates the set of frequency tables containing statistics (in the form of frequencies of character occurrences) for each element and non-enumerated attribute. The XML Parser is invoked a *second* time by the kernel to construct a tokenized form – tag, attribute, or data value – of the XML document. These tokens are supplied in streaming fashion to the kernel which calls for each token, based on its type, one of the following encoders:

Enum-Encoder is used for meta-data and enumerated-type data items. Each start-tag of an element is encoded by a '`T`' followed by a unique element-ID. All end-tags are encoded by '`/`'s. Attribute names are encoded by the character '`A`' followed by a unique attribute-ID. Enumerated-type attribute values, on the other hand, are encoded using the symbol table information.

Huffman-Compressor is used for non-enumerated data items. This module implements the non-adaptive Huffman coding compression scheme. It encodes each element/attribute value with the help of its associated Huffman tree, which is constructed from its corresponding frequency table. The last byte of the encoded sequence is padded to be *byte-aligned*, and this encoded sequence is then "escaped" so that the compressed XML document can be parsed without ambiguity.

The compressed output of the above encoders, along with the various frequency and symbol tables, is called the *Compressed Internal Representation* (CIR) of the compressor and is fed to **XML-Gen**, which converts the CIR into a semi-structured compressed XML document. This conversion is done *on the fly* during the second pass while the document is being compressed.

3.4. Compression Example

Consider an XML document fragment along with its DTD as shown in Figures 2 and 3, respectively. The document represents a student database with five elements: `STUDENT`, `NAME`, `YEAR`, `PROG` and `DEPT`. The `STUDENT` element has a `rollno` attribute, while `DEPT` has a `name` attribute of enumerated-type.

An abstract view of the compressed version of the above document is shown in Figure 4. Here, the tag `STUDENT` is encoded as `T0`, `NAME` as `T1`, `YEAR` as `T2`, `PROG` as `T3` and `DEPT` as `T4`. All end tags are encoded as '/'s. The attributes `rollno` and `name` are encoded as `A0` and `A1`, respectively. The unique element/attribute IDs and the encodings for the attribute `name` of `DEPT` element are determined

IEEE COMPUTER SOCIETY

```
<!- student.xml -->
<STUDENT rollno = "604100418">
   <NAME>Pankaj Tolani</NAME>
   <YEAR>2000</YEAR>
   <PROG>Master of Engineering</PROG>
   <DEPT name = "Computer_Science">
</STUDENT>
```

Figure 2. Fragment of the Student DB

```
<!- DTD for the Student database -->
<!ELEMENT STUDENT (NAME, YEAR, PROG, DEPT)>
<!ATTLIST STUDENT rollno CDATA ♯REQUIRED>
<!ELEMENT NAME (♯PCDATA)>
<!ELEMENT YEAR (♯PCDATA)>
<!ELEMENT PROG (♯PCDATA)>
<!ELEMENT DEPT EMPTY>
<!ATTLIST DEPT name (Computer_Science
   | Electrical_Engineering
       ⋮
   | Physics | Chemistry)
>
```

Figure 3. DTD for the Student DB

by the DTD parser in the first pass. *nahuff*(s) denotes the output of the Huffman-Compressor for an input data value s, while *enum*(s) denotes the output of the Enum-Encoder for an input data value s, which is an enumerated attribute. As is evident from Figure 4, the compressed document output in the second pass is semi-structured in nature, and maintains the property of validity with respect to the compressed DTD.

3.5. Query Processing

The compressed-domain *query processing engine* consists of a *lexical analyzer* that emits tokens for encoded tags, attributes, and data values, and a *parser* built on top of this lexical analyzer that does the matching and dumping of the matched "records" (which in the XML world are semi-structured tree fragments). As all the tokens are byte-aligned, the lexical analyzer that tokenizes the CIR is able to operate on a byte-by-byte basis. This means no bit-by-bit operations are necessary, considerably speeding up the lexical analysis. The parser, which makes a depth-first-search traversal of the XML document, maintains information about its current location (path) in the XML document and the contents of the set of XML nodes that it is currently processing.

For *exact-match* or *prefix-match* queries, the query path and the query predicate are converted to the compressed-domain equivalent. During parsing of the compressed XML document, when the parser detects that the current path matches the query path, and that the compressed data value matches the compressed query predicate, it outputs

```
T0 A0 nahuff(604100418)
   T1 nahuff(Pankaj Tolani) /
   T2 nahuff(2000) /
   T3 nahuff(Master of Engineering) /
   T4 A1 enum(Computer_Science) /
/
```

Figure 4. Abstract view of XGrind document

the matched XML fragment. Note that the compressed-domain pattern-match is byte-by-byte and not a bit-by-bit pattern-match, which would be highly inefficient. In fact, the matching requires significantly *less* work in the compressed domain, since the number of bytes to be processed have considerably decreased.

For *range* or *partial-match* queries, only the query path is compressed. While parsing the compressed XML document, when the parser detects that the current path matches the query path, the associated data value is decompressed and used for evaluating the match. This decompression is required since the compression scheme we use is not "order preserving" (i.e. given two strings s_1, s_2 and their respective compressed versions c_1, c_2, then $s_1 > s_2 \not\Rightarrow c_1 > c_2$). Only the records whose element/attribute values fall in the range are fully decompressed and returned to the user.

3.6. Implementation

We have implemented the XGrind tool in C/C++. The SAX API [21] XML Parser provided in [23] was used for implementing the XML Parser. Lex and Yacc were used for implementing the DTD Parser as well as the parser for the semi-structured compressed XML document. Also, we wrote our own non-adaptive Huffman-Compressor.

4. Experimental Framework

In this section, we describe the experimental setup used to profile XGrind's performance. We evaluated XGrind on a representative set of real and synthetic XML documents, including one generated from Xmark, the recently announced XML benchmark [24]. To our knowledge, there do not exist any prior queryable XML compressors. Therefore, we have attempted to place the XGrind performance results in perspective as follows: (a) For the compression ratio and compression time metrics, we compare with the XMill compressor; (b) For the query processing time metric, we compare against the Native query processor, which we built around XMill's XML parser and operates directly on the original uncompressed document. Our experiments were conducted on a PIII, 700 MHz machine, running Linux (TurboLinux 6.0), with 64 MB main memory and 18 GB local IDE disk.

IEEE COMPUTER SOCIETY

4.1. XML Documents

Document	S	R	D	E	A	N	M
xmark	1145	1	8	77	16	0	1
conferences	382	1.04M	3	25	5	0	10
journals	294	0.76M	3	15	2	0	11
shakespeare	161	740	6	22	0	0	22
ham-radio	361	0.70M	4	24	0	0	1
student1	960	5M	3	6	2	1	1
student4	1408	5M	3	7	5	4	1

Table 1. Document Statistics

The details of the XML documents considered in our study are summarized in Table 1. `S` (size) refers to the total disk space occupied by the document in MBs; `R` (records) indicates the number of top-level records in the document; `D` (depth) indicates the maximum level of nesting; `E` (elements), `A` (attributes) and `N` (enums) indicate the number of elements, attributes and enumerated-attributes, respectively, in the document; `M` (scale-up) indicates the number of times the original file has been concatenated.

The XML documents used in our study cover a variety of sizes, document characteristics and application domains, and are listed below:

xmark: This document was generated from Xmark, the xml-benchmark project, using their `xmlgen` data generator [24]. It models an auction database and is deeply-nested with a large number of elements and attributes. Many of the element values are long textual passages.

conferences, journals: These documents represent `conference` and `journal` entries, respectively, from the DBLP archive [20].

shakespeare: This document is the publicly available XML version of the plays of Shakespeare [22]. Similar to **xmark** above, many of the element values are long textual passages.

ham-radio: This document was obtained from the publicly available Ham Radio database of the US Government's Federal Communications Commission [15]. It has the highest percentage of meta-data content (approximately 70%) among the set of XML documents considered here.

student1: This is a synthetically generated XML document that represents a database of student information. The DTD for this document has one attribute – *name* (of the department) – which is an enumerated type.

student4: This is also a synthetically generated document, similar to *student1*, except that the DTD has *four* enumerated attributes – *year* (of registration), *name* (of the course), *name* (of the department), and *name* (of the previous school).

The reason that we have enlarged, by concatenation, some of the above documents is to ensure that our results scale to the large XML documents that are expected to be commonplace in the future, especially in the bioinformatics domain. We also ran our experiments on the original (unscaled) versions of these documents, and the results are consistent with those presented here.

4.2. XML Queries

We have evaluated query response times for a variety of *exact-match* and *range* queries, the details of which are given below (the queries are specified in XML-QL):

Exact-match queries: A sample exact-match query is shown in Figure 5. This query extracts the name of the student whose roll number (which is a "key" value) equals 123456789. We evaluated the query performance for randomly positioned records over the entire document and present here the results for the average case. For these queries, the parsers used in `XGrind` and `Native` were instrumented to stop when the desired record was found – that is, it is assumed that the search keys are *unique*.

```
CONSTRUCT <student rollno=$r> {
   WHERE
      <student rollno=123456789>
         <name>$n</name>
         <year>$y</year>
         <dept name=$d>
      </student> IN "student.xml",
   CONSTRUCT <name>$n</name>
} </student>
```

Figure 5. XML-QL exact-match query

Range queries: A sample range query is shown in Figure 6, which extracts all students whose date of joining is between the years 1998 and 2000. We evaluate a wide range of query selectivities in our experiments.

```
CONSTRUCT <student rollno=$r> {
   WHERE
      <student rollno=$r>
         <name>$n</name>
         <year>$y</year>
         <dept name=$d>
      </student> IN "student.xml",
      $y ≥ 1998 and $y ≤ 2000
   CONSTRUCT <name>$n</name>
} </student>
```

Figure 6. XML-QL range query

4.3. Compression Performance Metrics

From the compression perspective, we compare `XGrind`'s *compression ratios* and *compression times* with that of `XMill`. These metrics are defined below:

Compression Ratio (CR): Defined as $CR = 1 - \frac{sizeof(compressed\ file)}{sizeof(original\ file)}$.

1063-6382/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

Compression Ratio Factor (CRF): Normalizes the compression ratio of XGrind with respect to XMill, that is, $CRF = \frac{CR_{XG}}{CR_{XM}}$.

Compression Time (CT): Time taken to compress the XML file.

Compression Time Factor (CTF): Normalizes the compression time of XGrind with respect to XMill, that is, $CTF = \frac{CT_{XG}}{CT_{XM}}$.

4.4. Query Performance Metrics

From the query perspective, we compare XGrind's *query response times* with that of Native. These metrics are defined below:

Query Response Time (QRT): Total time required to execute the query.

Query Speedup Factor (QSF): Normalizes the query response time of Native with respect to XGrind, that is, $QSF = \frac{QRT_{Na}}{QRT_{XG}}$.

5. Performance Results

In this section, we present the performance results for the documents and queries described in the previous section. The results for the compression metrics are described first, followed by the results for the query metrics.

5.1. Compression Metrics

Document	CR_{XG}	CR_{XM}	CRF
xmark	55.03	70.95	0.78
conferences	57.44	84.61	0.68
journals	57.85	85.59	0.68
shakespeare	54.96	74.12	0.74
ham-radio	76.85	93.54	0.82
student1	77.13	91.74	0.84
student4	82.12	93.87	0.87
Average			**0.77**

Table 2. Comparison of compression ratios

The *compression ratio* statistics for the seven XML documents are shown in Table 2. We see here that, as expected, XGrind has lower compression ratio than XMill, but the important point is that its compression ratio factor (CRF) is, on the average, *about 77%* that of XMill. Also, the worst case is *within 68%* of XMill. These results were also true for a variety of other documents that we considered in our experiment evaluation.

Further, the results for student1 and student4 indicate that the compression ratio for XGrind *improves* with increase in the number of enumerated attributes. Experiments with other documents also showed similar results. Since we expect a significant usage of enumerated attributes in real life XML documents, XGrind's compression ratios will probably be better in practice than those shown here, that is, the values presented here are "conservative".

Document	CT_{XG}	CT_{XM}	CTF
xmark	1246	878	1.41
conferences	442	222	1.99
journals	344	170	2.02
shakespeare	183	125	1.46
ham-radio	353	182	1.93
student1	978	471	2.07
student4	1328	647	2.05
Average			**1.83**

Table 3. Comparison of compression times

The *compression time* statistics are shown in Table 3. We observe here that XGrind's compression time is always within about *twice* the time taken by XMill. This is not surprising since the XGrind compression scheme is two-pass, whereas XMill is one-pass. Further, for the *xmark* and *shakespeare* documents, which have longer text passages, the XGrind compression time is within about one and a half times the time taken by XMill. This is because XMill's pattern-based compression scheme turns out to be computationally costlier than the simple character-based encoding used in XGrind for such long text segments.

5.2. Query Metrics

Document	QRT_{XG}	QRT_{Na}	QSF
xmark	80	185	2.00
conferences	27	68	2.51
journals	21	53	2.52
shakespeare	14	31	2.21
ham-radio	20	73	3.65
student1	46	184	4.00
student4	50	250	5.00
Average			**3.12**

Table 4. Exact-Match Query Performance

Document	DT_{XM}	DT_{gzip}
xmark	663	488
conferences	151	145
journals	116	107
shakespeare	71	65
ham-radio	125	73
student1	288	336
student4	428	479

Table 5. Decompression Times

IEEE COMPUTER SOCIETY

We now move on to the query performance comparisons. For *exact-match* queries the average query response times are shown in Table 4. The inferences we make from the results are: First, $QRT_{XG} \ll QRT_{Na}$ in all the cases, and this is made explicit in the QSF column, which measures the relative speed up of `XGrind` w.r.t. `Native`. The minimum QSF for `XGrind` is about 2 times and is typically much higher, overall averaging around 3.

Second, QRT_{XG} (as well as QRT_{Na}) is much less than the time it takes `XMill` or `gzip` to decompress the XML document, shown as DT_{XM} and DT_{gzip}, respectively, in Table 5. This result has the important implication that `XGrind` would perform substantially better than `XMill` or `gzip` even if these tools were supplied with an algorithm that takes *zero time* to execute exact-match queries over an uncompressed XML document. Moreover, `XGrind` would require less space to process the query than `XMill` or `gzip`.

Document	Sel	QRT_{XG}	QRT_{Na}	QSF
conferences	1	71	136	1.92
	10	87	150	1.72
	50	153	205	1.34
journals	1	54	106	1.96
	10	64	117	1.83
	50	115	162	1.41
shakespeare	1	27	57	2.11
	10	35	66	1.89
	50	65	88	1.35
ham-radio	1	43	139	3.23
	10	58	150	2.59
	50	125	255	2.04
student1	1	138	364	2.64
	10	166	390	2.35
	50	292	540	1.85
student4	1	140	497	3.55
	10	172	549	3.19
	50	319	751	2.35

Table 6. Range Query Performance

Sel	Average QSF (over all documents)
1	2.56
10	2.26
50	1.72

Table 7. Range Query Average Performance

For *range* queries, the query response times for a spectrum of result selectivities (1%, 10%, and 50%) are shown in Table 6. The selectivity is evaluated with respect to the number of top-level nodes,[6] but it is straightforward to extend our experiments to lower-level nodes.

The results in Table 6 show that $QRT_{XG} \ll QRT_{Na}$ for all selectivities over all the documents. This is made explicit in the $Average\ QSF$ values shown in Table 7, which averages the performance for a selectivity across all documents. Note that for 1% and 10% selectivity, which are typically the types of queries seen in practice, the average improvement is above *2.25 times* with respect to `Native`. Further, even for a selectivity as coarse as 50 %, the improvement is by over 70 percent.

[6] Therefore, this experiment is not meaningful for *xmark* since it has only one top-level node.

5.3. Summary and Discussion

Our experimental results indicate that `XGrind` provides a reasonably good compression ratio – on the average, about three-quarters that of `XMill`, and always at least two-thirds that achieved by `XMill`. Further, the compression time is always within a factor of two of that of `XMill`. These numbers are especially encouraging given that we are (a) using element/attribute-granularity compression, rather than document-granularity compression, (b) using a simple character-based Huffman coding scheme, rather than a pattern-based approach, and (c) making two passes over the original XML document to provide context-free compression. Further, note that while compression is a "one-time" operation, querying is a repeated occurrence – therefore, any overheads in document compression time would be quickly amortized over large query sequences.

On the query processing front, `XGrind` provides substantially improved response times over `Native`. For an exact-match predicate on a key field, `XGrind` does better by a factor of three, on average. Similarly, even for range queries where a significant portion of the document would necessarily be decompressed, `XGrind`'s response time is about half that of `Native`, on average.

Finally, while `XGrind` exhibits a good performance profile in general, it performs particularly well with respect to all performance metrics when the XML document exhibits the following characteristics: (a) long textual passages, and (b) several enumerated-type attributes.

6. Related Work

On the research front, apart from XMill, there are two other XML compressors that we are aware of: Millau [4], which is designed for efficient encoding and streaming of XML structures; and a more recent encoding based on Prediction by Partial Match (PPM), called Multiplexed Hierarchical Modeling (MHM) [2]. Similarly, on the industrial front, there are quite a few companies – for example, `www.xmlzip.com`, `www.ictcompress.com` and `www.dbxml.com` – which supply XML compression products. However, a common feature of all these tools is that their focus is primarily on reducing the size of the compressed document, ignoring the issue of being query-friendly, which we consider here.

IEEE COMPUTER SOCIETY

7. Conclusions and Future Work

In this paper, we have considered, for the first time, the problem of developing XML compression algorithms that permit querying to be directly carried out on the compressed document. To this end, we developed an prototype tool called `XGrind`, which is built around non-adaptive Huffman coding that supports context-free decompression at the token granularity. `XGrind` also has a special encoder for enumerated types, a frequent occurrence in XML documents. The most novel feature of `XGrind`, however, is that the compressed document retains exactly the same semi-structured layout as the original document. This facilitates the use of similar parsing techniques for both versions. More importantly, it permits us to build indexes directly on the compressed document, which we expect to be a major value-addition in practice. Finally, an attractive side-effect of XGrind's token-granularity, context-free, compression scheme is that the compressed XML document is more robust with regard to transmission and disk errors as compared to `XMill` or `gzip`.

We evaluated `XGrind`'s query performance against `Native` and the results indicate substantially improved query response times. These benefits are obtained while simultaneously and efficiently achieving compression ratios that are comparable with that of `XMill`. To further improve the performance and utility of the `XGrind` tool, we could:

- identify "fixed-schema" elements (i.e. no {*/+/?} modifiers for the nested elements) from the DTD, and not repeat their schema in the compressed document.
- use sampling in the statistics gathering phase to reduce document compression times.
- use the statistics gathering phase to identify the enumerated-type elements/attributes if the document happens to not have a DTD.

Acknowledgments

We are very grateful to Aditya Nori for his technical inputs and programming support during the early stages of this work. J. R. Haritsa was supported in part by a research grant from the Dept. of Bio-technology, Govt. of India.

References

[1] T. Bray, et al. "Extensible Markup Language (XML) 1.0", October 2000, *http://www.w3.org/TR/REC-xml*.

[2] J. Cheney, "Compressing XML with Multiplexed Hierarchical PPM Models", *Proc. of IEEE Data Compression Conf.*, May 2000.

[3] A. Deutsch, et al. "A Query Language for XML", June 2001, *http://www.w3.org/TR/xquery/*.

[4] G. Girardot and N. Sundaresam, "Millau: an encoding format for efficient representation and exchange of XML over the Web", *http://www9.org/w9cdrom/154/154.html*.

[5] G. Graefe, "Options in Physical Database", *ACM SIGMOD Record*, September 1993.

[6] G. Graefe and L. Shapiro, "Data Compression and Database Performance", *Proc. of ACM/IEEE CS Symp. on Applied Computing*, April 1991.

[7] J. Hopcroft and J. Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley, 1979.

[8] D. Huffman, "A Method for Construction of Minimum-Redundancy Codes", *Proc. of IRE*, September 1952.

[9] B. Iyer and D. Wilhite, "Data Compression Support in Databases", *Proc. of VLDB*, September 1994.

[10] H. Liefke and D. Suciu, "XMill: An Efficient Compressor for XML Data", *Proc. of ACM SIGMOD*, May 2000.

[11] H. Liefke and D. Suciu, "XMill: An Efficient Compressor for XML Data", *Tech. Rep. MS-CIS-99-26*, Dept. of Computer and Information Science, Univ. of Pennsylvania, October 1999.

[12] J. McHugh, et al. "Indexing Semi-structured Data", *Technical Report*, Computer Science Dept., Stanford University, January 1998.

[13] G. Ray, J. Haritsa and S. Seshadri, "Database Compression: A Performance Enhancement Tool", *Proc. of 7th Intl. Conf. on Management of Data (COMAD)*, December 1995.

[14] I. Witten, R. Neal and J. Cleary, "Arithmetic Coding For Data Compression", *Comm. of ACM*, June 1987.

[15] *ftp://ftp.ictcompress.com/pub/xmltestfiles*

[16] *http://www.ebi.ac.uk*

[17] *http://www.gzip.org*

[18] *http://www.gzip.org/algorithm.txt*

[19] *http://www.ictcompress.com/xml.html*

[20] *http://www.informatik.uni-trier.de/~ley/db*

[21] *http://www.megginson.com/SAX*

[22] *http://www.oasis-open.org/cover/bosakShakespeare200.html*

[23] *http://www.research.att.com/sw/tools/xmill*

[24] *http://www.xml-benchmark.org*

1063-6382/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY