XGRIND: A Query-friendly XML Compressor

Pankaj M. Tolani Jayant R. Haritsa*

Database Systems Lab, SERC

Indian Institute of Science

Bangalore 560012, India

Abstract

XML documents are inherently 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

In recent years, 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. The size increase is estimated to be as much as 400 percent [10]!

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

`gzip` [6], and thereby reduce the size of the document. An alternative approach is to design an XML-specific compressor. This approach was used in the `XMill` tool, proposed recently by Liefke and Suciu [12]. `XMill` achieves compression ratios typically in excess of 80 percent on large XML documents by grouping semantically related data items into "containers" and compressing each container separately with a specialized compressor that is ideal for that container (optional), followed by a `gzip` on that container. For example, the meta-data (in the form of XML tags and attributes) and the data (element and attribute values) are compressed separately. A performance study [12, 13] showed `XMill` to consistently provide improved compression ratios as compared to `gzip`.

Since `XMill` is designed to minimize the *size* of the equivalent compressed XML, it is attractive in terms of reducing network bandwidth requirements for transmission of XML documents, and disk space requirements for storage of XML documents. 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 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 [23, 28, 29, 31, 33], 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] 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 [12].

Based on the above observations, we propose in this paper a new compression tool called **XGrind**, that directly support queries in the compressed domain. That is, instead of compressing at the granularity of the entire document, it compresses at the granularity of individual element/attribute values using a context-free compression scheme based on Huffman encoding [7]. 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 document 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* [37] 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 number of XML documents. For example, major repositories of genomic data such as the European Bioinformatics Institute (EBI) [39], 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.

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 according to the recently announced XML benchmark [36]. Our study considers a variety of metrics including the compression ratio, the compression time, and the query processing times. Since, to our knowledge, there do not exist any prior queryable XML compressors, we have attempted to place the `XGrind` performance results in perspective by comparing it against the following yardsticks: (a) `XMill`, with regard to the compression ratio and compression time

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

with regard to the query processing time metric.

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/Arithmetic encoding scheme, rather than a pattern-based approach, and (c) making two passes over the original XML document to provide context-free compression.

We hasten to mention that the initial pass of gathering the statistics for context-free compression could be optimized by sampling in case of huge XML documents. This would marginally reduce the compression ratios and reduce the compression times considerably. The implementation details of the sampling pass need to be worked out.

Further, note that while compression is a "one-time" operation, querying would probably be a repeated occurrence – therefore, any overheads in document compression time would quickly be amortized over large query sequences.

On the query processing front, XGrind provides substantially improved query processing times as compared to Native for a variety of common XML-QL [2] queries. For example, 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.

In summary, we present here a new compression tool for XML documents that is "query-friendly", making it practical to simultaneously achieve both reasonable compression and good query performance. To the best of our knowledge, this is the first quantitative work on queryable XML compression.

1.3 Organization

The remainder of this paper is organized as follows: Background material on compression techniques and the XMill compressor is provided in Section 2. The architectural design and implementation details of our new XGrind compressor are presented in Section 3. The performance model and the experimental results are highlighted in Sections 4 and 5, respectively. Section 6 presents the results of the XGrind tool using the Arithmetic-Compressor module as against the Huffman-Compressor module used for the results shown in Section 5. Related work on XML compression is overviewed in Section 7. Finally, in Section 8, we summarize the conclusions of our study and outline future avenues to explore.

In this section, we overview background material on compression techniques, and also the `XMill` compressor. We restrict our attention to lossless compression techniques in this paper since we expect that for XML document databases, which store mainly textual and numeric data, only such techniques can be used.

Most lossless data compression techniques are based on one of two models: **statistical** or **pattern**.[3] In statistical modeling, each distinct *character* of the input data is encoded, with the code assignment being 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.

Yet 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 techniques are based on one of the following algorithms: **Huffman**, **Arithmetic**, **LZ77** or **LZ78**. The Huffman coding 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. We describe the Huffman, Arithmetic, and LZ77 techniques here.

2.1 Huffman Coding

In Huffman coding [7], 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 (also known as prefix-free encoding). 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 the path from the root to the leaf node corresponding to that character in the Huffman tree.

As mentioned earlier, both adaptive and non-adaptive versions of Huffman coding exist. In non-adaptive Huffman coding, the Huffman tree is completely built before encoding starts, using the known frequency distribution of the characters in the data to be compressed. The tree remains unchanged for the entire duration of the encoding process. The decoder builds the same tree using the saved frequency distribution

[3] An exception is the classical Run-length encoding scheme which simply recognizes successive repetitions of characters.

with a Huffman tree that is built using an *assumed* frequency distribution of the characters in the input data. A common practice is to assume that all characters are equally likely to occur. As the encoding process proceeds and more data is scanned, the Huffman tree is modified based on the data seen up to that point. Therefore, the Huffman tree changes dynamically during the encoding phase and the same character can have different codes depending on its position in the data being compressed (unlike non-adaptive Huffman).

2.2 Arithmetic Coding

A limitation of Huffman coding is that each character is encoded into an *integral* number of bits. This means that the codes may often be longer than that strictly required for the character. For example, a character with probability of occurrence 0.9 can be coded minimally in 0.135 bits (from information-theoretic considerations [4]), but requires 1 full bit in this scheme.

Arithmetic coding attempts to address the above shortcoming of Huffman coding. Here, the compressed version of the input data is represented by the interval between two real numbers of *arbitrary* precision, (x, y), where $0 \leq x < y \leq 1$. At the start, the range is initialized to the entire interval [0,1), and this range is progressively refined. During the encoding process, each character is assigned an interval within the current range, the width of the interval being proportional to the probability of occurrence of that character. The range is then narrowed to that portion of the current range which is allocated to this character. So, as encoding proceeds and more data is scanned, the interval needed to represent the data becomes smaller and smaller, and the number of bits needed to specify the interval grows. The more likely characters reduce the range less than the unlikely characters and hence add fewer bits to the compressed data. The implementation details of this scheme are given in [8, 26].

Arithmetic coding also has adaptive and non-adaptive versions, in exactly the same manner as that described previously for Huffman coding.

2.3 LZ77 Coding

The LZ77 coding [9] 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

[4] Information content of a character is its *entropy*= $-p_c log_2 p_c$, where p_c is the probability of its occurrence.

the same string table as the encoder and constructs it incrementally in a similar manner.

2.4 The XMill Compressor

The `XMill` [12, 13] compressor represents, as mentioned in the Introduction, 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.

For the above kind of documents, `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 content 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 (see e.g.[31, 34]). 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.

To implement the above control flow, the XML document is parsed by a hand-crafted SAX (Simple API for XML) parser [35] that sends a stream of tokens to a path processor, which assigns each token to an appropriate container, and containers are then compressed independently with their associated compression mechanism.

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

In this section, we first describe the design features of our new XML compressor, XGrind, which are intended to ensure both good query performance and reasonable compression ratios. We then present its 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:

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 its unique element-id. Each end-tag is encoded by a '/'. Each attribute name is encoded by the character 'A' followed by its unique attribute-id.

3.1.2 Enumerated-type Attribute Value Compression

Enumerated-type attribute values are a common occurence in XML documents. For example, the states of a country, or the set of departments in a company, or the set of zipcodes, are all instances of frequently occuring 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 and 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 and 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 point of occurrence 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 searching for occurrences of its corresponding encoded sequence in the compressed document.

to a data item is dependent on the entire contents of the document prior to the occurence of the data item. That is, only with a complete decompression of the prior contents is it possible to decode a given sequence. On the other hand, context-free coding of strings is possible with the *non-adaptive* versions of compression algorithms like Huffman coding and Arithmetic coding. To support the non-adaptive feature, two passes have to be made over the input XML document, as discussed in the previous section: 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 zipcodes 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 level 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. To be more precise, *exact-match* and *prefix-match* user queries can be *completely* carried out directly on the compressed document, while *range* or *partial-match* 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 output compressed 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 [37], 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.

[5] This is similar to collecting column or domain statistics for compression in an RDBMS [34].

the XGrind encoding scheme for the meta-data and enumerated-type attribute values. Let $h_{\mathcal{X}}(\mathcal{D})$ denote the compressed DTD and $h_{\mathcal{X}}(\mathcal{X})$ denote the compressed XML document. The following property is a consequence of the "context freeness" of the compression scheme and the semi-structured nature of the output.

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

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 under inverse homomorphisms [11].

3.3 System Architecture

The architecture of the `XGrind` compressor along with the information flows is shown in Figure 1. The **XGrind kernel** module is the heart of the compressor. It starts off by invoking the **DTD Parser**, which parses the DTD of the input 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 which contain statistics (in the form of frequencies of character occurrences) for each element and non-enumerated attribute. The kernel then invokes the XML Parser for a *second* time to construct a tokenized form – tag, attribute, or data value – of the XML document. The 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 module if the token type is meta-data or an enumerated-type data item. Each start-tag of an element is encoded by a '`T`' followed by its unique element-id. Each end-tag is encoded by a '`/`'. Each attribute name is encoded by the character '`A`' followed by its unique attribute-id. As mentioned earlier, this coding scheme is similar to that used by `XMill`. Enumerated-type attribute values are coded using the symbol table information.

Huffman-Compressor module [6]

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 the assoiciated frequency tables. The last byte of the encoded sequence is *padded* with `'0's (bits)`, and *escaped* so that the compressed XML document can be parsed

[6] We also have the Aritchmetic-Compressor module that implements the non-adaptive Arithmetic coding compression scheme. We do not mention the Arithmetic-Compressor details in the text to avoid repetition.

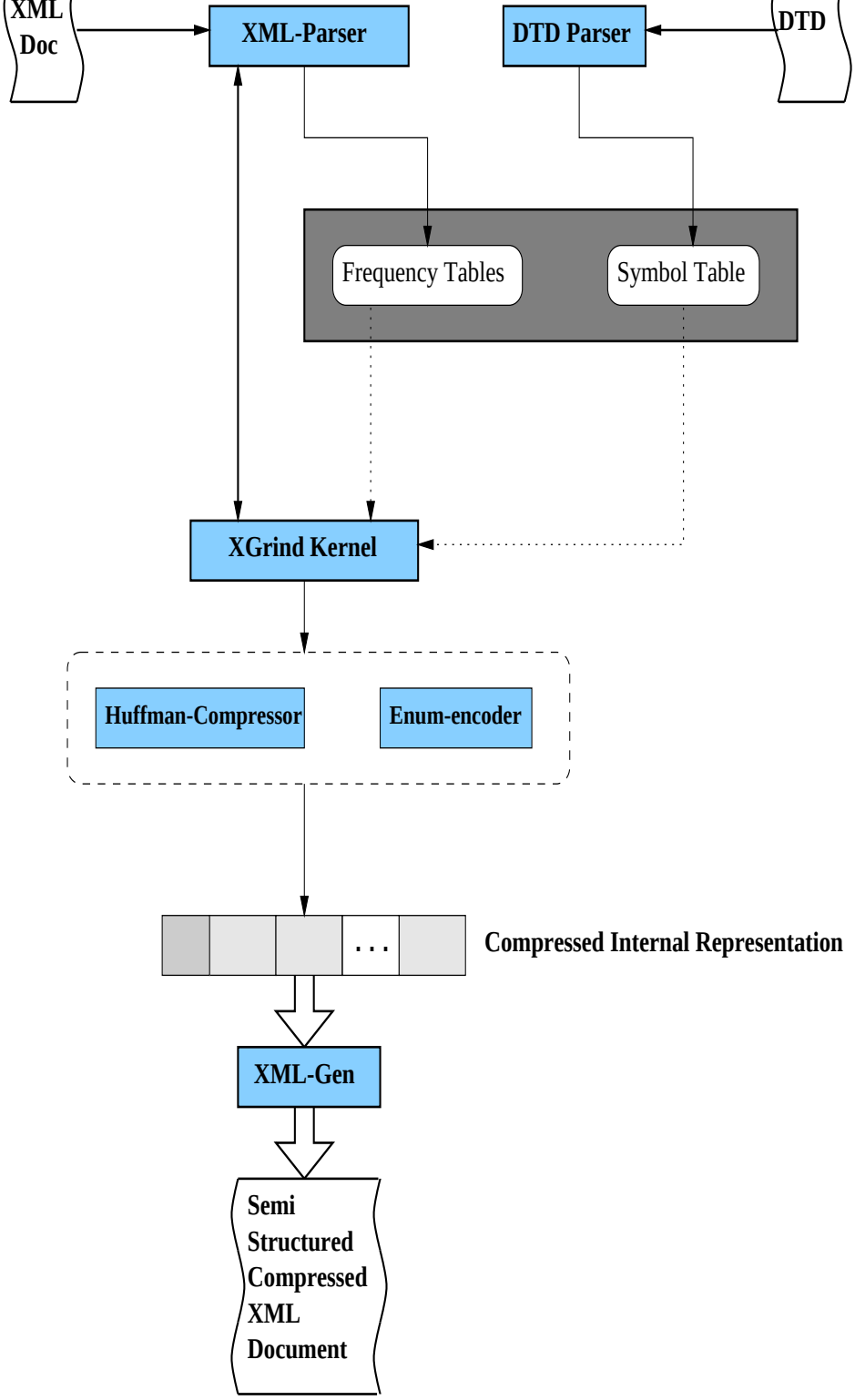

Figure 1: **Architecture of `XGrind` Compressor**

```
<!- student.xml -->
<STUDENT rollno = "604100418">
   <NAME>Pankaj Tolani</NAME>
   <YEAR>2000</YEAR>
   <PROG>M.E</PROG>
   <DEPT name = "Computer_Science_and_Automation"/>
</STUDENT>
```

Figure 2: **An XML fragment of the Student database**

without ambiguity. Escaping is done so that the demilter does not appear in the compressed text and hence it can be extracted and decompressed 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 the **XML-Gen** module, 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

We now demonstrate the working of `XGrind` with an 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. [7]

For the above document, the XGrind Kernel module invokes the DTD Parser module to examine the DTD for enumerated-type attributes and stores their values in its symbol table. It also initializes the frequncy tables for the elements and non-enumerated type attibutes in the XML document. In our example, the attribute `name` along with its list of values is inserted in the symbol table. The XGrind Kernel module next calls the XML Parser which scans the input XML document and collects the statistics for each element and attribute type. A second scan of the input XML document is now made and a stream of tokens is returned to the kernel during this pass. Depending on the type of the token, the appropriate compressor for it is called. That is, for all the tags, attributes, and enumerated-type attribute values, the Enum-encoder module is invoked, while the Huffman-Compressor module is invoked for the rest of the data items. The

[7] The underscores in the enumerated values in Figure 3 are necessary as the current XML standard [1] does not allow white space as a valid character in an enumerated string.

```
<!- 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_and_Automation
   | Electrical_Communication_Engineering
   | Electrical_Engineering
      ⋮
   | Supercomputer_Education_And_Research_Centre)
>
```

Figure 3: **DTD for the Student database**

```
T0 A0 nahuff(604100418)
   T1 nahuff(Pankaj Tolani) /
   T2 nahuff(2000) /
   T3 nahuff(M.E) /
   T4 A1 enum(Computer_Science_and_Automation) /
/
```

Figure 4: **Abstract view of the compressed XML document**

output of this together with all the meta-tables (that is, the symbol table along with the frequency tables containing the statistics) forms the CIR. This is given as input to the XML-Gen module which outputs the final semi-structured compressed document. An abstract view of the output compressed 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 '/'. The attributes `rollno` and `name` are encoded as `A0` and `A1`, respectively. *nahuff*`(s)` denotes the output of the Huffman-Compressor module for an input data value `s`, while *enum*`(s)` denotes the output of the Enum-encoder module for an input data value `s`, which is an enumerated attribute.

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, would be 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. At the time of parsing the compressed-domain equivalent, when the current path matches the query path, and the compressed data value matches the compressed query predicate, the parser outputs the matched record and halts. Note that the compressed-domain pattern-match is also a byte-by-byte comparison, and not a bit-by-bit pattern-match, which would be highly inefficient. In fact, the matching requires much less work in the compressed domain, as the number of bytes are much fewer in the compressed version.

- For range or partial-match queries, only the query path is compressed. At the time of parsing the compressed-domain equivalent, when the current path matches the query path, the data value is decompressed and used for evaluating the match. This decompression is required since compression scheme we use is not "order preserving" (i.e. given two strings s_1, s_2 and their respective code words 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++. The SAX API [35] XML Parser provided in [14] was used for implementing the XML Parser module, while Lex and Yacc were used for implementing the DTD Parser module. We wrote our own non-adaptive Huffman-Compressor module and parser for the semi-structured compressed XML document (again, using Lex and Yacc).

In this section, we describe our experimental setup to evaluate `XGrind`. We evaluated `XGrind` on a representative set of real and synthetic XML documents, including one generated according to the recently announced XML benchmark [36]. Since, to our knowledge, there do not exist any prior queryable XML compressors, we have attempted to place the `XGrind` performance results in perspective by comparing it against the following yardsticks: (a) `XMill`, with regard to the compression ratio and compression time metrics, and (b) `Native`, a parser which directly operates on the original uncompressed XML document, with regard to the query processing time metric. 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.

4.1 XML Documents

XML document	Size	Records	Scaleup	Depth	Elems	Attrs	Enums
xmlbenchmark	1.119 GB	1	1	8	77	16	0
conferences	382 MB	1.04M	10	3	25	5	0
journals	294 MB	0.76M	11	3	15	2	0
shakespeare	161 MB	740	22	6	22	0	0
xpress	361 MB	0.70M	1	4	24	0	0
student1	960 MB	5M	1	3	6	2	1
student4	1.375 GB	5M	1	3	7	5	4

Table 1: **Statistics of the data sets**

The details of the XML documents considered in our study are summarized in Table 1. The `size` field refers to the total disk space occupied by the document; the `records` field indicates the number of top-level records in the document; the `scaleup` field indicates the number of times the original file has been concatenated; the `depth` field indicates the maximum level of nesting; the `elems`, `attrs` and `enums` fields indicate the number of elements, attributes and enumerated-attributes, respectively, in the document.

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

xmlbenchmark: This document was generated using the recently announced XML benchmark data generator, `xmlgen` [36]. It is deeply-nested and has a large number of elements and attributes. Here, element values are often long textual passages.

archive [40]. Since DTDs are not available for these documents, we created the DTDs ourselves. However, we hasten to add that this is not strictly required since the `XGrind` tool works *without a DTD* also – the only impact is that the compression times will increase marginally, because, without the DTD, internal frequency statistics hash tables will have to be created on-the-fly during the initial pass, as the XML document is being parsed.

shakespeare: This document is the publicly available XML version of the plays of Shakespeare [41]. Here, element values are often long textual passages.

xpress: This document is obtained from the publicly available Ham Radio database of the US Government's Federal Communications Commission [42]. This *real dataset* has the highest percentage of meta-data content (70%) amongst the set of XML documents we consider 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. The student names (an element in this XML document with PCDATA content) are generated by concatenating words from the *ispell* spellchecker dictionary, and are not just random text.

student4: This is also a synthetically generated XML 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 scaled up (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 bio-informatics domain. We hasten to add that we also ran our experiments on the original(unscaled) versions of these documents, and the results are consistent with those presented here. Therefore, the scale-up does not prejudice our numbers in anyway.

Sample fragments of the XML documents are shown in the appendix.

4.2 Performance Metrics

Compression Metrics:

On the compression side, we compare `XGrind`'s *compression ratios* and *compression times* with that of `XMill`. These metrics are listed below:

Compression Ratio (CR) : This is defined as $CR = \frac{sizeof(compressed\ file)}{sizeof(original\ file)}$.

Compression Ratio Factor (CRF) : This represents the compression ratio of `XGrind` normalized to that of `XMill` and is defined as $CRF = \frac{CR_{XGrind}}{CR_{XMill}}$.

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

Compression Time Factor (CTF) : This represents the compression time of `XGrind` normalized to that of `XMill` and is defined as $CTF = \frac{CT_{XGrind}}{CT_{XMill}}$.

Query Metrics:

On the query side, we compare `XGrind`'s *query response times* with that of `Native`. The comparison metrics are listed below:

Query Response Time (QRT) : This is total time required to execute a user query (in seconds).

Query Speedup Factor (QSF) : This represents the query response time speedup obtained by `XGrind` normalized to that of `Native` and is defined as $QSF = \frac{QRT_{Native}}{QRT_{XGrind}}$.

4.3 XML Queries

For evaluating query response times, we use a representative subset from the "Ten Essential XML queries" described in [3]. These queries are described below, and are presented using the XML-QL query language constructs [3].

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 the results here for the average case. For these queries, the parsers used in `XGrind` and `Native` were instrumented to stop when the desired pattern was found.

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 2001. We evaluate a wide range of query selectivities in our experiments.

Delete queries : This query deletes the record of the student whose roll number (which is a "key" value) equals 123456789. Again, we evaluated the query performance for randomly positioned records over the entire document and present the results here for the average case.

```
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**

```
CONSTRUCT <student rollno=$r> {
    WHERE
        <student rollno=$r>
            <name>$n</name>
            <year>$y</year>
            <dept name=$d>
        </student> IN "student.xml",
        $y ≥ 1998 and $y ≤ 2001
    CONSTRUCT <name>$n</name>
} </student>
```

Figure 6: **XML-QL range query**

5 Results

In this section, we present the results for the data sets and queries described in the previous section. We present the results for the compression metrics first, followed by the query metrics. The results shown here are for the Huffman-Compressor module. We club the results for the Arithmetic-Compressor module in Section 6 to avoid confusion. Also these performance numbers are not of much interest, as the query response times are much higher for the marginal improvements in compression numbers.

Document	$\mathbf{CR_{XGrind}}$	$\mathbf{CR_{XMill}}$	CRF
xmlbenchmark	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
xpress	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**

5.1 Compression Metrics

5.1.1 Compression Ratio

The compression ratios (CRs) for `XGrind` and `XMill` and the associated compression ratio factors (CRFs) for the seven XML documents are shown in Table 2. Based on these statistics, we make the following observations:

- As expected, `XGrind` has lower compression ratio than `XMill`, but the important point is that its CRF is, on the average, *about 77%* that of `XMill`. Also, the worst case is within 68% of `XMill`. These results were also also true for a variety of other documents that we considered (but are not described here due to space limitations) in our experiment evaluation.

- The results for `student1` and `student4` show 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".

5.1.2 Compression Time

The compression time performance of `XGrind` is shown in Table 3, and compared with that of `XMill`. From the values in the table, we see that `XGrind`'s compression time is always within about *twice* the time taken by `XMill`. This is not surprising, as the `XGrind` compression scheme is two-pass, as against the one-pass compression scheme used in `XMill`. Also, for the *xmlbenchmark* and *shakespeare* datasets, which

Document	$\mathbf{CT_{XGrind}}$ (in secs)	$\mathbf{CT_{XMill}}$ (in secs)	CTF
xmlbenchmark	1246	878	1.41
conferences	442	222	1.99
journals	344	170	2.02
shakespeare	183	125	1.46
xpress	353	182	1.93
student1	978	471	2.07
student4	1328	647	2.05
Average			**1.83**

Table 3: **Comparison of compression times**

have longer text passages, the `XGrind` compression time is within about one and a half times the time taken by `XMill`. This is because the `XMill` pattern-based compression scheme turns out to be computationally costlier than the simple character-based encoding used in `XGrind` for such long text segments.

Finally, note that while document compression is usually a "one-time" operation, querying the document is a repeated occurrence – therefore, the extra compression time overhead of `XGrind` will be quickly amortized from its benefits with regard to query performance, described next.

5.2 Query Metrics

We now move on to the query perfomance comparisons. Here, we compare the QRTs of `XGrind` (QRT_{XGrind}) with that of `Native` (QRT_{Native}), for exact-match, range-match and delete queries, respectively.

5.2.1 Exact-Match query

For exact-match queries, a sample of which was presented in Figure 5, the average query response times are shown in Table 4. The inferences we make from the results are the following:

- First, $QRT_{XGrind} \ll QRT_{Native}$ 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 for `XGrind` is about 2 times and is typically much higher, overall averaging around 3.

- Second, QRT_{XGrind} (as well as QRT_{Native}) is much less than the time it takes `XMill` or `gzip` to decompress the XML document, as shown in Table 5. So, if a tool like `XMill` or `gzip` were to be

Document	$\mathbf{QRT_{XGrind}(in\ secs)}$	$\mathbf{QRT_{Native}(in\ secs)}$	QSF
xmlbenchmark	80	185	2.00
conferences	27	68	2.51
journals	21	53	2.52
shakespeare	14	31	2.21
xpress	20	73	3.65
student1	46	184	4.00
student4	50	250	5.00
Average			**3.12**

Table 4: **Exact-Match Query Performance**

Document	$\mathbf{XMill\ decompression\ time(in\ secs)}$	$\mathbf{gzip\ decompression\ time(in\ secs)}$
xmlbenchmark	663	488
conferences	151	145
journals	116	107
shakespeare	71	65
xpress	125	73
student1	288	336
student4	428	479

Table 5: **Decompression Times**

used for compression, and even if there were an algorithm that takes *zero time* to execute exact-match queries over an uncompressed XML document, QRT_{XGrind} would still perform substantially better than `XMill` or `gzip`. Further, `XGrind` would also require less space to process the query than `XMill` or `gzip`.

5.2.2 Range query

For range queries, a sample of which was presented in Figure 6, the query response times for a spectrum of answer selectivities (1%, 10%, and 50%) are shown in Table 6. Here, we consider selectivity with respect to the number of top-level nodes. Hence, this experiment is not meaningful for *xmlbenchmark*, since it has only one top-level node. However, it is straightforward to extend our experiments to lower-level nodes.

The inferences we make from the results in Table 6 are the following:

Document	%Selectivity	$\mathbf{QRT_{XGrind}(in\ secs)}$	$\mathbf{QRT_{Native}(in\ secs)}$	QSF
xmlbenchmark	-	-	-	-
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
xpress	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**

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

Table 7: **Range Query Average Performance**

- $QRT_{XGrind} \ll QRT_{Native}$ 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 given selectivity across all the 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* w.r.t. `Native`. Further, even for a selectivity as coarse as 50 %, the improvement is by over 70 percent.

Document	$\mathbf{QRT_{XGrind}(in\ secs)}$	$\mathbf{QRT_{Native}(in\ secs)}$	QSF
xmlbenchmark	318	743	2.33
conferences	109	285	2.61
journals	84	284	2.61
shakespeare	45	107	2.37
xpress	62	266	4.29
student1	174	783	4.50
student4	191	1035	5.41
Average			**3.44**

Table 8: **Delete Query Performance**

For delete queries, the average query response times are shown in Table 8. The inferences we make from the results are the following:

- $QRT_{XGrind} \ll QRT_{Native}$ for all the documents and this is made explicit in the QSF columns, showing a minimum speedup for `XGrind` of about 2 times and typically much higher, overall averaging about 3.5.

6 Results using Arithmetic-Compressor

Here, we present the results for the same data sets and queries for the `XGrind` tool using the Arithmetic-Compressor module and compare these qualitatively with those for the `XGrind` tool using the Huffman-Compressor module.

6.1 Compression Metrics

6.1.1 Compression Ratio

The compression ratios (CRs) for `XGrind` and `XMill` and the associated compression ratio factors (CRFs) for the seven XML documents are shown in Table 9. Based on these statistics, and by comparing these with the results of the `XGrind` tool using the Huffman-Compressor module, we observe that there is just a marginal increase in compression ratios in all cases. This is due to the fact that we use element/attribute granularity compression, and in most cases the number of bits saved in Arithmetic compression as against

Document	$\mathbf{CR_{XGrind}}$	$\mathbf{CR_{XMill}}$	CRF
xmlbenchmark	55.0	70.95	0.78
conferences	57.6	84.61	0.68
journals	58.0	85.59	0.68
shakespeare	55.0	74.12	0.74
xpress	76.9	93.54	0.82
student1	77.5	91.74	0.84
student4	82.4	93.87	0.87
Average			**0.77**

Table 9: **Comparison of compression ratios**

Huffman compression make up the last byte of the compressed text, which is finally padded with '0' bits (in both cases) before writing in the CIR. Note, the padding is done for efficient tokenizing of the CIR at decompression/query time using byte operations.

6.1.2 Compression Time

Document	$\mathbf{CT_{XGrind}}$ (in secs)	$\mathbf{CT_{XMill}}$ (in secs)	CTF
xmlbenchmark	1800	878	2.05
conferences	973	222	4.38
journals	745	170	4.38
shakespeare	441	125	3.52
xpress	418	182	2.29
student1	1438	471	3.09
student4	2000	647	3.09
Average			**3.25**

Table 10: **Comparison of compression times**

The compression time performance of `XGrind` is shown in Table 10, and compared with that of `XMill`. Comparing CTFs with that for Huffman-Compressor based `XGrind` compressor suggest that the compression times are much higher, for a marginal increase in compression ratios shown earlier in this section.

We now move on to the query perfomance comparisons. Here, we compare the QRTs of `XGrind` (QRT_{XGrind}) with that of `Native` (QRT_{Native}), for exact-match and range-match queries, respectively.

6.2.1 Exact-Match query

Document	$\mathbf{QRT_{XGrind}(in\ secs)}$	$\mathbf{QRT_{Native}(in\ secs)}$	QSF
xmlbenchmark	80	185	2.00
conferences	26	68	2.61
journals	20	53	2.65
shakespeare	13	31	2.38
xpress	20	73	3.65
student1	46	184	4.00
student4	50	250	5.00
Average			**3.18**

Table 11: **Exact-Match Query Performance**

For exact-match queries, the query response times using the Arithmetic-Compressor are similar to that using the Huffman-Compressor module. This is because decompression (which is costlier for Arithmetic Compression as compared to Huffman compression) is done only for one record qualifying the exact-match query predicate. Apart from decompression, the work done in both cases is almost the same (most of which is comparing bytes in the compressed-domain).

6.2.2 Range query

The performance for range queries using the Arithmetic-Compressor module is much worse compared to those using the Huffman-Compressor module. The query response times are more for `XGrind` than `Native` in most cases. This is because the Arithemetic-Compressor module is much slower than the Huffman-Compressor module, as the computations in the implementation of rithmetic compression are much more involved than that in the implementation of Huffman compression.

Document	%Selectivity	$\mathbf{QRT_{XGrind}(in\ secs)}$	$\mathbf{QRT_{Native}(in\ secs)}$	QSF
xmlbenchmark	-	-	-	-
conferences	1	172	136	0.79
	10	222	150	0.67
	50	459	205	0.44
journals	1	130	106	0.81
	10	165	117	0.70
	50	345	162	0.46
shakespeare	1	62	57	0.91
	10	110	66	0.60
	50	195	88	0.45
xpress	1	102	139	1.36
	10	150	150	1.00
	50	375	255	0.68
student1	1	340	364	1.07
	10	420	390	0.92
	50	876	540	0.61
student4	1	345	497	1.44
	10	435	549	1.26
	50	957	751	0.78

Table 12: **Range Query Performance**

%Selectivity	Average QSF(over all documents)
1	1.06
10	0.85
50	0.57

Table 13: **Range Query Average Performance**

7 Related Work

Given the tremendous spurt in popularity of XML over the last two years, it is not surprising that compression of these verbose documents has also started attracting attention. On the research front, apart from XMill, the only other compression tool that we are aware of is Millau [15], which is designed for efficient encoding and streaming of XML structures. This tool is designed for *small* XML

few companies – for example, `http://www.xmlzip.com`, `http://www.ictcompress.com` and `http://www.dbxml.com` – which claim to have XML compression tools. As mentioned in [12], the `xmlzip` tool from `http://www.xmlzip.com` runs out of memory on large documents. The tool from `http://www.ictcompress.com` claims to provide significantly more compression than XMill, but they do not consider the issue of being query-friendly.

8 Conclusions

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 algorithm called `XGrind`, which is built around a non-adaptive Huffman encoder 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 its guarantee 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. Another nice *side-effect* of XGrind's token-granularity, context-free, compression scheme is that the compressed XML document is more *tolerant* to transmission / disk errors as compared to `XMill` or `gzip`, and hence the compression scheme is more *robust*. We evaluated `XGrind`'s query performance against `Native` and the results indicate substantially improved query response times. Further, these benefits were obtained while simultaneously achieving compression ratios that are comparable with that of `XMill`.

Next, we plan to investigate the following interesting issues:

- Using the information in the DTD corresponding to the XML document, it is possible to determine the elements that have a fixed-schema without any {*/+/?} modifiers for the nested elements. This makes it possible to avoid repeating the schema in the compressed document for fixed-schema elements. We expect that this will improve the compression ratios of `XGrind` even further, without having any retrogade effect on its query performance.

- The recently-proposed XML Schema [4] exposes the data types of the elements. We could use this information to employ specialized context-free compressors for specific data types.

- The initial pass of gathering the statistics for context-free compression could be optimized by sampling in case of huge XML documents. This would marginally reduce the compression ratios and

be worked out.

References

[1] T. Bray, J. Paoli and C. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0", *http://www.w3.org/TR/1998/REC-xml-19980219*, February 1998.

[2] A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu, "A Query Language for XML", *Proc. of Intl. World Wide Web Conf.* , 1999.

[3] M. Fernandez, J. Simeon and P. Wadler, "XML Query Languages: Experiences and Exemplars", *http://www-db.research.bell-labs.com/user/simeon/xgquery.html*, 2000.

[4] *http://www.w3.org/XML/Schema*

[5] H. K. Reghbati. "An Overview of Compression Techniques", *IEEE Computer*, April 1981.

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

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

[8] I. H. Witten, et al. "Arithmetic Coding For Data Compression", *Comm. of ACM*, June 1987.

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

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

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

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

[13] 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.

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

[15] 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*,

[16] S. Eggers, F. Olken, and A. Shoshani. "A Compression Technique for Large Statistical databases", *Proc. of VLDB Conf.*, September 1981.

[17] J. McCarthy. "Metadata Management for Large Statistical Databases", *Proc. of VLDB Conf.*, September 1982.

[18] G. D. Severance. "A Practitioner's Guide to Database Compression – A Tutorial", *Information Systems*, January 1983.

[19] M. Bassiouni and K. Hazboun. "Utilization of Character Reference Locality for Efficient Storage of Databases", *Proc. of 2nd Intl. Workshop on Statistical Database Management*, September 1983.

[20] E. Lefons, A. Silvestri, and F. Tangorra. "An Analytic Approach to Statistical Databases", *Proc. of VLDB Conf.*, October 1983.

[21] D. Batory. "Index Coding: A Compression Technique for Large Statistical Databases", *Proc. of 2nd Intl. Workshop on Statistical Database Management*, September 1983.

[22] T. A. Welch. "A Technique for High Performance Data Compression", *IEEE Computer*, June 1984.

[23] M. A. Bassiouni. "Data Compression in Scientific and Statistical Databases", *IEEE Trans. on Software Eng.*, October 1985.

[24] G. V. Cormack. "Data Compression in Database Systems", *Comm. of ACM*, December 1985.

[25] M. J. Carey and M. Livny. "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication", *Proc. of VLDB Conf.*, August 1988.

[26] D. W. Jones. "Application of Splay Trees to Data Compression", *Comm. of ACM*, August 1988.

[27] H. Korth, and A. Silberschatz. *Database System Concepts*, 2nd ed., McGraw-Hill, 1991.

Science Symp. on Applied Computing, April 1991.

[29] G. Graefe. “Options in Physical Database”, *SIGMOD Record*, September 1993.

[30] M. A. Roth and S. J. Van Horn. “Database Compression”, *SIGMOD Record*, September 1993.

[31] B. R. Iyer and D. Wilhite. “Data Compression Support in Databases”, *Proc. of VLDB Conf.*, September 1994.

[32] A. Zandi, B. Iyer, and G. Langdon. “Sort Order Preserving Data Compression for Extended Alphabets”, *Data Compression Conf.*, 1993.

[33] G. Ray. “Data Compression in Databases”, *Master’s Thesis*, Dept. of Computer Science and Automation, Indian Institute of Science, June 1995.

[34] 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.

[35] “Simple API for XML”, *http://www.megginson.com/SAX/*.

[36] A.R. Schmidt, F. Waas, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey, R.Busse “The XML Benchmark Project”, April 2001. *http://monetdb.cwi.nl/xml/Benchmark/benchmark.html*

[37] J. McHugh, J. Widom, S. Abiteboul, Q. Luo and A. Rajaraman, “Indexing Semistructured Data”, *Technical Report*, Stanford University, January 1998.

[38] *http://www.expasy.ch/sprot*.

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

[40] *http://www.informatik.uni-trier.de/∼ley/db*.

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

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