XCQ: XML Compression and Querying System

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

Peter T. Wood   Mark Levene
School of Computer Science and Information Systems
Birkbeck College, University of London
{ptw, mark}@dcs.bbk.ac.uk

ABSTRACT

We present our development of an XML compression and querying tool, which is called *XML Compression and Querying System* (XCQ). This system is developed based on a novel technique called *DTD Tree and SAX Event Stream Parsing* (DSP). This technique is designed for efficient compression of XML documents that conform to a given DTD without involving user expertise. A reasonable compression ratio, which is comparable to that of XMill, is achieved by DSP. The compressed documents in XCQ adopt a *partitioned path-based data grouping* which supports evaluating queries without running a full decompression. We demonstrate with examples how to query compressed documents in XCQ.

Keywords

XML, compression, querying, SAX parsing

1. INTRODUCTION

We have developed a system called XCQ. It is used for compressing XML documents that conform to a given DTD. It also supports querying compressed documents without fully decompressing them. This system consists of two major parts. They are the *Compression Engine* and the *Querying Engine*. In the following sections we present a brief overview of these engines.

2. THE XCQ COMPRESSION ENGINE

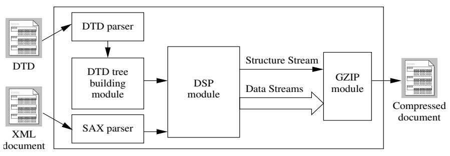

Figure 1: The Compression Engine Architecture

Figure 1 shows the architecture of the Compression Engine. This architecture is used for realizing the DSP technique. The ideas used in DSP are (1) to extract the *structural information* [2, 3] from the input XML document that cannot be inferred from a given DTD during the parsing process, and (2) to group the *data elements* in the document based on their corresponding *tree paths* in the *DTD tree*. The structural information here refers to the values of operator occurrences such as repetition operators (* and +), optional operators (?) and child indices of decision nodes (|). Data elements here refer to attribute and PCDATA values within the document. In DSP, two objects of a SAX event stream [8] and a tree data structure are adopted to model a given XML document and a given DTD, respectively. The generation of a SAX event stream is done by the XML parser whereas the creation of a DTD tree is done by the DTD tree building module which utilizes the outputs from the DTD parser (as depicted in Figure 1). For example, a DTD tree built based on the DTD given in Figure 2 is shown in Figure 3. This technique solves the long compression time and large memory consumption problems in [2, 3], since XML data is converted into a SAX event stream and only a small portion of the stream is resident in the main memory of the engine at run-time.

```
<!ELEMENT library (entry*)>
<!ELEMENT entry (author, title, year, publisher?, (course_note|paper|book), num_copy)>
<!ELEMENT author EMPTY>
<!ATTLIST author name CDATA>
<!ELEMENT title (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT course_note EMPTY>
<!ELEMENT paper EMPTY>
<!ELEMENT book EMPTY>
<!ELEMENT num_copy (#PCDATA)>
```

Figure 2: A DTD for a library XML document

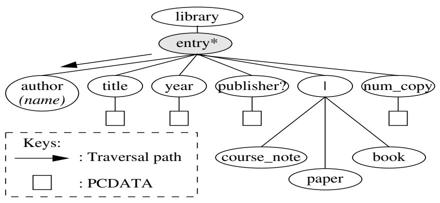

Figure 3: The DTD tree corresponding to the DTD in Figure 2

The generated SAX event stream and the created DTD tree are passed to the DSP module which performs structural information extraction and data elements grouping. The module uses a pseudo-depth first (PDF) traversal to traverse the DTD tree with respect to the SAX event stream in order to explore the required information. PDF traversal means that the module performs a depth-first traversal until an operator node or a choice node is encountered. The module then determines which of the traversal paths is followed in the next step. For operator nodes, a bit is used to indicate if the next SAX event token matches the current DTD tree node. The bit also tells the decoder whether the module has traversed the subtree of the operator node in the current parsing state. For choice nodes, each edge to a corresponding child is labelled with an index starting from 0. A byte is used to indicate which of the child nodes matches the next SAX event token. The byte value also shows the path that the module has taken to traverse the tree.

For example, assume the module encountered the "entry*" node in the DTD tree (the shaded node in Figure 3). If the next incoming SAX event token is an "entry" element (i.e. a match), the module assigns a 1-bit for this operator occurrence and follows the depth first traversal path (see the arrow in Figure 3) to traverse the subtree. Otherwise, a 0-bit is assigned and the subtree is not traversed. So if there are 1000 entries in the XML document, 1000 1-bits and 1 0-bit are assigned for the “entry*” node occurrences.

Now, assume the module encountered the “|” node. If the next incoming SAX event is a book element (a match with the third child), a byte with value 2 is assigned to the “|” occurrence which corresponds to the SAX event that matches the third child of the node. The module then picks the third path to traverse in the subtree. The module outputs the extracted structural information to the *Structure Stream*.

When the module encounters a data element, it checks the current DTD tree parsing state to extract the path that addresses the element. The module then outputs the data element to its corresponding *Data Stream* according to the extracted path. This grouping approach helps a generic text compressor to achieve a higher compression ratio [4]. In addition to the path-based grouping, XCQ also partitions each data stream into blocks[1], which is illustrated in Figure 4. Each block contains a certain number of elements under the same path. This partition approach to storing compressed data degrades the compression rate, but it improves the query response time. We discuss this issue in the next section.

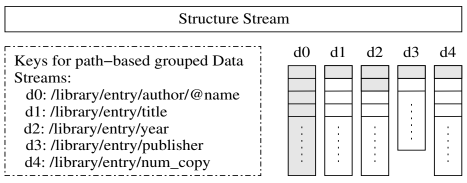
Structure Stream
Keys for path−based grouped Data Streams:
d0: /library/entry/author/@name
d1: /library/entry/title
d2: /library/entry/year
d3: /library/entry/publisher
d4: /library/entry/num_copy
d0
d1
d2
d3
d4

Figure 4: Partitioned Path-Based Grouping

The Structure Stream and the blocks in the Data Streams are then compressed individually using a text compressor, such as gzip [6]. These compressed components are then packed in a single file as output by the Compression Engine. In our experiments, we find that a better compression ratio than XMill [4] can be achieved at the expense of compression time, if no partitioning is made on the data streams in XCQ. XCQ also achieves, in general, a better compression ratio and compression time than XGrind [5], a known XML compression system that supports querying compressed XML documents. Further improving the compression ratio could be done if the bzip2 [7] compressor is used in the Compression Engine. However, preliminary results indicate that this would substantially increase the compression and query response times.

3. THE XCQ QUERYING ENGINE

XCQ supports querying compressed documents by only partially decompressing them. We demonstrate this by the following three queries formulated in XPath [1]:

Q1. entry[author/@name=“Tom” and publisher=“Clear Ltd.”]
Q2. sum(//num_copy)
Q3. count(//entry)

In Q1, two data groups are involved during query evaluation. The output returns those XML fragments under matched “entry” elements. We first explain how to evaluate the expression “author/@name=“Tom”” in Q1. Since there is no indexing on the compressed data, the engine needs to decompress the whole data group *d0* and to test each record in the group against the value “Tom”. Assuming there are two records, record 9 and record 214, fulfilling the first expression, the engine then needs to find the corresponding “publisher” records for evaluating the second expression[2].

To find the corresponding record indices, the engine parses the structure stream against the DTD tree and retrieves the record indexes in all data groups which correspond to the matched record indexes in *d0*. Assuming record 4 in block 1 and record 167 in block 2 of data group *d3* are the corresponding records to the candidate results returned from the first expression, the engine just needs to decompress block 1 and block 2 of *d3* and retrieves the two matched records for evaluating the second expression.

Assume that only record 4 fulfills the second expression. The engine then decompresses the corresponding blocks (assuming also all of them are in block 1) and retrieves the corresponding records, which are found during the structure stream parsing, in other data groups to construct the result. When processing Q1, we decompress only the shaded blocks in Figure 4.

In Q2, only one data group is involved and the result of the query is an aggregation value. In this case, the engine just needs to decompress the data group *d4* and sum up all the data values in that group. Structure stream parsing for finding corresponding records in other data groups is not needed.

We can see that a smaller block size (i.e. using a finer block partitioning) helps the querying performance, since a more precise portion of the compressed document is decompressed during query evaluation. However, this degrades the compression ratio, since fewer redundancies in the data streams can be eliminated by a text compressor if each block is compressed as a finer individual unit.

Queries involving only the structure of the document can be answered without decompressing the Data Streams. For example, to answer Q3, XCQ only needs to parse the structure stream against the DTD tree once and returns the number of “entry*” node occurrences that are assigned a 1-bit.

4. CONCLUSIONS

In this poster, we presented the development of a prototype XCQ, which is based on the technique we have called DSP. Without involving user expertise, XCQ enables users to efficiently compress XML documents, which conform to a given DTD, with comparable compression ratios to state of the art systems. XCQ also supports querying over partially decompressed documents, an area which merits further investigation.

REFERENCES

[1] XML Path Language (XPath) 1.0, W3C Recommendation 1999, http://www.w3.org/TR/xpath/

[2] M. Levene and P. T. Wood. *XML Structure Compression*, Proc. of the 2nd Int. Workshop on Web Dynamics, 2002.

[3] N. Sundaresan and R. Moussa. *Algorithms and Programming Models for Efficient Representation of XML for Internet Applications*. Proc. of the 10th Int. WWW Conf., p. 366-375, 2001

[4] H. Liefke and D. Suciu. *XMill: an efficient compressor for XML Data*. Proc. of ACM SIGMOD Conf. on Management of Data, p. 153-164, 2000

[5] P. M. Tolani and J. R. Haritsa. *XGRIND: A Query-friendly XML Compressor*. IEEE Proc. of the 18th Int. Conf. on Data Engineering 2002

[6] J. Gailly and M. Adler. *gzip 1.2.4* . http://www.gzip.org/

[7] J. Seward. *bzip2 1.0.2*. http://sources.redhat.com/bzip2/

[8] SAX - http://www.saxproject.org/

[1]The meaning of the shaded blocks is discussed in Section 3

[2]The number of elements in group *d3* is fewer since they are optional elements