Under consideration for publication in Knowledge and Information Systems

XCQ: A Queriable XML Compression System

Wilfred Ng[1], Wai-Yeung Lam[1], Peter T. Wood[2] and Mark Levene[2]

[1]Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong; [2]School of Computer Science and Information Systems, Birkbeck, University of London, Malet Street, London, UK

Abstract. XML has already become the de facto standard for specifying and exchanging data on the Web. However, XML is by nature verbose and thus XML documents are usually large in size, a factor that hinders its practical usage, since it substantially increases the costs of storing, processing, and exchanging data. In order to tackle this problem, many XML-specific compression systems, such as XMill, XGrind, XMLPPM, and Millau, have recently been proposed. However, these systems usually suffer from the following two inadequacies: they either sacrifice performance in terms of compression ratio and execution time in order to support a limited range of queries, or perform full decompression prior to processing queries over compressed documents.

In this paper, we address the above problems by exploiting the information provided by a Document Type Definition (DTD) associated with an XML document. We show that a DTD is able to facilitate better compression as well as generate more usable compressed data to support querying. We present the architecture of the XCQ, which is a compression and querying tool for handling XML data. XCQ is based on a novel technique we have developed called *DTD Tree and SAX Event Stream Parsing* (DSP). The documents compressed by XCQ are stored in *Partitioned Path-Based Grouping* (PPG) data streams, which are equipped with a *Block Statistics Signature* (BSS) indexing scheme. The indexed PPG data streams support the processing of XML queries that involve selection and aggregation, without the need for full decompression. In order to study the compression performance of XCQ, we carry out comprehensive experiments over a set of XML benchmark datasets.

Keywords: XML; Document Type Definitions; Compression algorithms; Query Processing; Performance

1. Introduction

The Extensible Markup Language (XML) (Bray et al, 2004) is proposed under the auspices of the World Wide Web Consortium (W3C) as a standardized data format designed

Received Nov 22, 2004
Revised Jul 20, 2005
Accepted Sep 12, 2005

for specifying and exchanging data on the Web. With the proliferation of mobile devices, such as pocket PCs, sensor networks and mobile phones, as a means of communication in recent years, it is reasonable to expect that in the foreseeable future a massive amount of XML data will be generated and exchanged between applications in order to perform dynamic computations over the Web.

When XML first emerged, the verbosity of XML markup was not considered a pressing issue from a design perspective. However, in practice XML documents are usually extremely large in size, due to the fact that they often contain much redundant data, such as repeated tags (for example, see the DBLP documents (Ley, 2005)). As a result, an XML-ized document is usually much larger than one conveying the same information but adopting a standard document format. For example, an XML-ized Weblog document in (Liefke and Suciu, 2000) is roughly three times the size of the original file.

Let us call this document size inflation the *inflation problem* of XML. The inflation problem seriously hinders the future use of XML in exchanging, parsing, and querying data, due to the fact that the data size grows much faster than the communication bandwidth. On the one hand, we enjoy the flexibility of XML, since the markup facilities of XML are intuitive for people and better able to facilitate web data exchange. On the other hand, we have to pay the extra cost of consuming more storage space and computational resources to store and process XML data.

In recent years, many XML-specific compression systems have been proposed and developed (Arion et al, 2004; Buneman et al, 2003; Cheney, 2002; Levene and Wood, 2002; Liefke and Suciu, 2000; Min et al, 2003; Sundaresan and Moussa, 2001; Tolani and Haritsa, 2002). However, these systems either do not make effective use of the information provided by a Document Type Definition (DTD) associated with an XML document (Arion et al, 2004; Buneman et al, 2003; Cheney, 2002; Liefke and Suciu, 2000; Min et al, 2003; Tolani and Haritsa, 2002), or do not support the querying of compressed documents directly (Cheney, 2002; Levene and Wood, 2002; Liefke and Suciu, 2000; Sundaresan and Moussa, 2001).

We believe that a DTD can improve the compression ratio of an XML document and help to produce more usable compressed data. For example, XMill (Liefke and Suciu, 2000) needs to perform a *full* decompression prior to processing queries over compressed documents, resulting in a heavy burden on system resources such as CPU processing time and memory consumption. At the other extreme, some technologies can avoid (full) XML data decompression but unfortunately only at the expense of compression performance. For example, XGrind (Tolani and Haritsa, 2002) adopts a homomorphic transformation strategy to transform XML data into a specialized compressed format and support direct querying on compressed data but at the expense of the compression ratio; thus the inflation problem is not satisfactorily resolved.

In this paper, we present our development of a prototype called the *XML Compression and Querying System* (XCQ), which attempts to balance the objectives of tackling the inflation problem and supporting querying on compressed data without the burden of performing a full decompression. We develop the XCQ prototype and study the feasibility of using XCQ in practice. We evaluate the performance of XCQ in compression, and demonstrate that a competitive compression ratio, which is comparable to that of XMill (Liefke and Suciu, 2000), is achieved by XCQ at the expense of compression time.

The underlying idea behind XCQ is to compress XML documents that conform to a DTD by making use of the DTD information to aid the compression process. We achieve this using our *DTD Tree and SAX Event Stream Parsing* (DSP) technique. In addition, we propose and analyze two simple but effective techniques for handling queries over compressed XML data. First, we use a novel Partitioned Path-Based Grouping

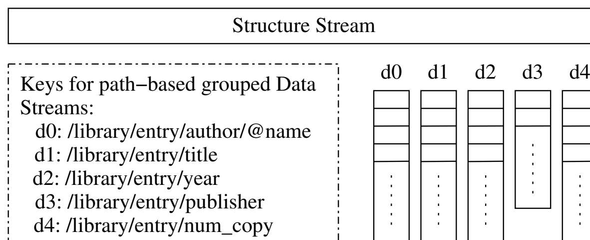


Fig. 1. Data Streams Partitioned using Path-Based Grouping (PPG)

(PPG) strategy for storing path-based compressed XML data in a number of streams of blocks. Second, we impose a minimal indexing scheme, called a *Block Statistics Signature* (BSS), on the compressed data blocks. We show that these techniques are not only efficient enough to support selection and aggregation queries over compressed XML data via *partial* decompression, but they also require a low computation and storage space overhead.

The main idea of our proposed PPG strategy is depicted in Figure 1, in which the structure stream, derived from a given XML document and its DTD, is stored and compressed separately from the data streams. XMill (Liefke and Suciu, 2000) also divides an XML document into a number of separate containers, one for the structure and one for each attribute and element name used in the document. The PPG scheme differs from this in at least two ways: firstly, the structure stream is encoded using information from the DTD; secondly, the data streams are based on paths in the DTD (as shown in Figure 1) rather than simply names.

PPG assumes that the DTD is non-recursive and hence can be represented as a DTD tree such as that shown in Figure 2, where `name` is an *attribute node*; `author`, `title`, `year`, `publisher`, and `num_copy` are *element nodes*; and `paper`, `course_note`, and `book` are *empty* elements. (A full description of DTD trees is given in Section 3.) The values in a data stream all have the same path back to the root of the DTD tree (or root of the XML document), as suggested in Figures 1 and 2.

Each PPG data stream is partitioned into its set of data blocks having a pre-defined *block size*, which helps to increase the overall compression ratio (cf. (Iyer and Wilhite, 1994; Liefke and Suciu, 2000; Ng and Ravishankar, 1997; Poess and Potapov, 2003)). A data block in a PPG data stream is able to be compressed or decompressed as an individual unit. This partitioning strategy allows us to access the data in a compressed document by decompressing only those data blocks that contain the data elements relevant to the input query, which we call the strategy of *partial decompression*.

For example, if the block size has n records per block, the first batch of n records in the stream d_0 are packed in the first data block, while the next batch of n records are packed in the second block. As such, each data block in the data streams contains a certain number of elements in the order listed in their corresponding data stream.

The data blocks in the data streams are compressed individually using the low-level compressor gzip (Gailly and Adler, 2003a). Intuitively, a smaller block size (i.e. using a finer partitioning in a PPG data stream) improves query performance, since a more precise portion of the compressed document can be identified for decompression in order to evaluate a query. However, there is a trade-off in that a finer block partitioning degrades the compression ratio, since fewer redundancies in the data streams can be eliminated by gzip if each block is compressed as a finer individual block unit.

The BSS indexing scheme is employed to aid the block retrieval in PPG data streams

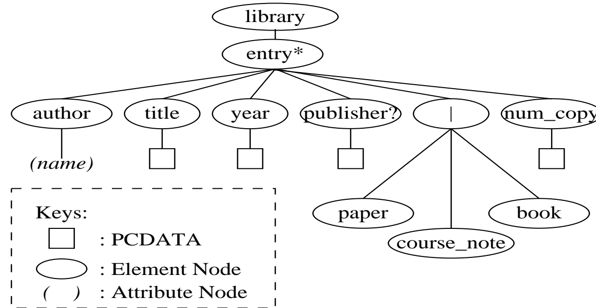


Fig. 2. A Library DTD Tree

when processing queries on XCQ compressed data. A BSS stream is suitable for block-oriented compressed data, which includes parameters such as *min* and *max* generated for each data block. This signature summarizes the content in a compressed data block, which supports more effective 'hitting' of the target blocks in answering queries. The storage space overhead required by the BSS indexing scheme is low. We do not need to generate a bit pattern for each record in a data block as some compressors do. With respect to the computation overhead, the operations of generating and scanning a signature can be carried out in $O(n)$ time. In the case of signature generation, n is the number of elements in a PPG data stream. In the case of signature scanning, n is the number of compressed data blocks in a PPG data stream.

The remainder of the paper is organized as follows. In Section 2, we explain the architecture of XCQ, which supports querying compressed documents. In Section 3, we present in detail the DSP technique, outline its working principles, and discuss the parsing algorithm that realizes the technique. In Section 4, we discuss the PPG strategy and the BSS indexing scheme used in XCQ. In Section 5, we present the experimental results of compressing real world XML documents using XCQ, which compare with a range of state-of-the-art compressors. In Section 6, we review related work and recent developments in XML compression. Finally, in Section 7, we give our concluding remarks and discuss future research work pertaining to XCQ.

2. XCQ Architecture

In this section, we present the architecture of XCQ and discuss how the XCQ system supports querying over partially decompressed documents. The architecture of XCQ comprises the *Compression Engine* and the *Querying Engine*. This prototype is developed using C++, the SAX XML parser of (Clark, 2004) and the gzip library of (Gailly and Adler, 2003b).

Figure 3 shows the architecture of the Compression Engine, which consists of the following components: the *DTD parser*, the *DTD tree building module*, the *SAX parser*, the *DSP module*, the *partition and indexing module*, and the *compression module*.

We now briefly explain the functionality of these modules below.

– *DTD Parser and DTD Tree Building modules*.
The DTD parser module parses the input DTD document and analyses its content. The result is utilized by the DTD Tree Building Module to construct a DTD tree. Elements that are used in multiple content models in the DTD are represented by

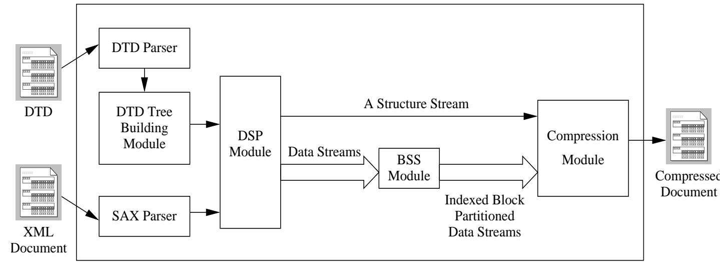


Fig. 3. The Compression Engine Architecture

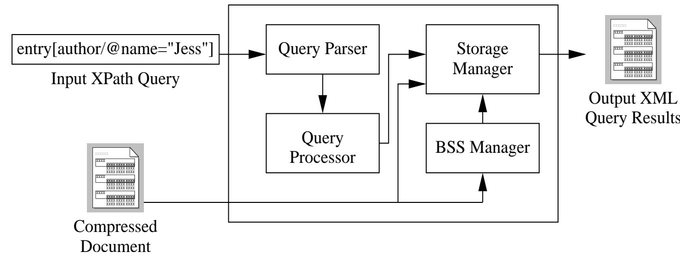


Fig. 4. The Architecture of the XCQ Querying Engine

separate nodes in the tree. The Tree Building Module assumes that the DTD is non-recursive.

– *SAX parser*.
This module uses the SAX parser in (Clark, 2004) in order to generate a SAX event stream (Megginson, 2004) that corresponds to the given XML document.
– *DSP module*.
This module implements the DSP algorithm, which we discuss in Section 3. It takes as input the DTD tree and the SAX event stream created by the DTD Tree Building module and the SAX Parser module, respectively. It outputs the corresponding structure stream and set of data streams.
– *Partition and indexing (BSS) module*.
This module first partitions the incoming data streams, generated by the DSP module, into their corresponding sets of blocks. Each block contains a certain number of data elements belonging to its corresponding PPG data stream. The module then generates a BSS index for each data block in a PPG data stream.
– *Compression module*.
The structure stream generated by the DSP module and the indexed PPG data streams generated by the partition and indexing module are compressed indivually and then packed and merged into a single file. The compression module also manages the data buffer in order to minimize disk access frequency. This module is built on top of the gzip compression libraries (Gailly and Adler, 2003b). We could have used the bzip2 library of (Seward, 2005) as an alternative in this module. However, we found that, in general, bzip2 substantially increases both the compression and query response times.

XCQ supports querying of compressed documents that conform to the *Partitioned Path-Based Data Grouping* by only partially decompressing them. The underlying idea

used in the engine is that it only decompresses portions of the compressed document that are relevant to the query evaluation. Figure 4 shows the architecture of the Querying Engine, which comprises the *query parser*, the *query processor* and the *storage manager*. We now briefly explain the functionality of each of these components below.

- *Query parser*.
  The query parser converts a query formulated in XPath (Clark and DeRose, 1999) into an internal form used in XCQ. A set of tokens are used to describe the content, such as the predicates and the relevant elements, of the input query. The tokenized query is then fed into the query processor.
- *Query processor*.
  The query processor is used to generate a set of *access commands* based on the input tokenized query. These access commands are used to instruct the storage manager to access the required portions of the compressed file. The query processor then evaluates the input query based on the results returned from the storage manager. The result generated by the query processor is passed back to the storage manager, which outputs the result as an XML document.
- *BSS manager*.
  The BSS manager is responsible for checking the data block signatures (i.e. BSS indexes). When the engine is initialized, the BSS manager loads the BSS indexing information from the header of the input compressed document into main memory. It subsequently helps the storage manager to determine whether a compressed data block contains the required data elements based on its BSS index.
- *Storage manager*.
  The storage manager is responsible for instructing the operating system to access the compressed files. It also provides buffer management to minimize disk I/O overhead.

Although we have concentrated on the compression and querying aspects of XCQ above, we should mention that any XML document compressed with XCQ can be faithfully recovered by decompression (except possibly for those whitespace characters that are not significant). Since both the structure stream and the data streams are compressed using gzip which is lossless, they can be recovered. The structure of the document can be reconstructed from the structure stream, as shown, for example, in (Levene and Wood, 2002). Finally, since the data streams in XCQ are written out in document order, it is straightforward to reconstruct the original XML document from the structure, the data streams and the DTD.

3. DTD Tree and SAX Event Stream Parsing

In this section, we first give an overview of the DTD Tree and SAX Event Stream Parsing (DSP) technique and highlight those of its features that are desirable for XML compression. We then present the DSP algorithm and illustrate the idea with a detailed example.

3.1. Overview of DSP

In DSP, we use a SAX event stream (Megginson, 2004) and a DTD tree data structure together to model a given XML document. The generation of a SAX event stream is carried out by the XML parser in (Clark, 2004), whereas the creation of a DTD tree is

```
<!ELEMENT library (entry*)>
<!ELEMENT entry (author, title, year, publisher?,
                        (paper|course_note|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)>
```

Fig. 5. A DTD for a Library XML Document

carried out by the DTD tree building module. We now illustrate our basic ideas about building a DTD tree.

In Figure 5, we show a simple DTD for an XML document of library information. A DTD tree for the document is built as follows. Each element that has a unary operator ("?", "$*$", or "$+$") applied to it, such as "entry" and "publisher", is transformed into a corresponding operator node, with the element operand as part of the node, as shown in Figure 2. This is a shorthand representation for an operator node with a single child. Sequences of elements (separated by the "," operator) are also represented implicitly by the ordering of child elements in the tree (as in Figure 2) unless nodes corresponding to the "," operator are required because of the complexity of the content model defined in the DTD.

Elements that comprise a set of alternatives, such as "(paper | course_note | book)", are transformed into a choice node with the corresponding elements as children. If the operands of "|" operator are expressions that are more complicated than a single element name, then more elaborate subtrees are built. An element having attributes, such as "author", is transformed into a tree node with the attributes associated with the node. PCDATA nodes are attached to those elements that are defined as "#PCDATA" in the DTD, such as "title". The generated DTD tree is as shown in Figure 2.

The DTD tree and the SAX event stream are processed by a module that implements DSP as shown in Figure 6. The functions of this module are (1) to extract the *structural information* (Levene and Wood, 2002; Sundaresan and Moussa, 2001) from the input XML document that cannot be inferred from the DTD during the parsing process, and (2) to group *data elements* in the document based on their corresponding *tree paths* in the DTD tree. By structural information we mean the information necessary to reconstruct the tree structure of the XML document. By data elements we mean the attributes and PCDATA within the document. The module parses the DTD tree by using a special traversal sequence, which depends on the SAX event stream in order to explore the required information. The output of this module is a stream of structural information, which we call the *structure stream*, and streams of XML data, which we call the *data streams*.

DSP possesses a number of desirable features for XML data compression, detailed as follows.

1. *Less memory is required.*
   Since an XML document is converted into a SAX event stream instead of a tree data structure, main memory usage of the compression engine is significantly reduced. In addition, our compressor can start the compression process once the DTD tree, which is usually very small in size, has been constructed. As a result, the thrashing problem

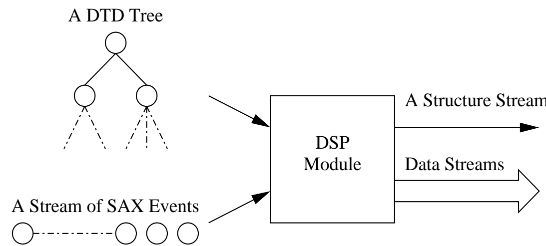


Fig. 6. The DSP Module in XCQ

is avoided and the compression time is shortened, which makes our approach more efficient in terms of time and space than the pure DDT (Sundaresan and Moussa, 2001) and SCA (Levene and Wood, 2002) approaches.

2. *Partial decompression is supported.*
   Data values with related semantics that have the same tree path in the DTD tree are grouped together in the same data stream. It has been shown in (Iyer and Wilhite, 1994; Liefke and Suciu, 2000; Sundaresan and Moussa, 2001) that data grouping assists generic text compressors to explore data redundancies among data and thus helps to increase the overall compression ratio. In addition, the indexed PPG compressed format supports querying over compressed data by performing partial decompression.
3. *No user expertise is required.*
   The streams output from the DSP module can be efficiently compressed without involving user expertise. Our compressor has the advantage over some unqueriable compressors such as XMill in that it does not require complex command line hints that describe the structure of the compressed document in order to specify the data grouping. In XCQ, such information is extracted from the DTD in an automated manner.

3.2. DSP Algorithm

The DSP algorithm is implemented in the DSP module. It is used for realizing a Pseudo-Depth First (PDF) traversal strategy, which explores the required information from the DTD tree and SAX event streams, in order to generate the structure and data streams.

The PDF traversal strategy varies from the conventional depth-first traversal when traversing the DTD tree. In particular, the traversal path is determined *on the fly* based on the input SAX event stream. Using PDF traversal, the DSP module traverses the DTD tree in a depth-first traversal manner with respect to the input SAX event stream *until* an operator node or a choice node is encountered. It then determines the subsequent traversal path based on what node in the DTD tree the next SAX event matches.

We now present the details of the DSP algorithm. Let the current node in the DTD tree be denoted by v.

1. If v is a PCDATA node, the DSP module first computes the path from the root node of the DTD tree to v, and then outputs the current SAX event to the data stream corresponding to this path. Finally, it moves to the node following v in depth-first order and waits for the next relevant SAX event (either a start-element event or a PCDATA event).

```
<library>
  <entry>
    <author name="Tom"/>
    <title>
      Comp123: Operating System: Introduction
    </title>
    <year>2003</year>
    <course_note/>
    <num_copy>3</num_copy>
  </entry>
  <entry>
    <author name="Jess Chu"/>
    <title>A Better World</title>
    <year>1874</year>
    <publisher>Clear LTD.</publisher>
    <book/>
    <num_copy>2</num_copy>
  </entry>
      ⋮
</library>
```

Fig. 7. A Simple XML Document Conforming to the DTD Given in Figure 5

2. If v is an element node, the module process the attributes returned by the current SAX event, waits for the next relevant SAX event and then moves to the node following v in depth-first order. For simplicity, we assume that all attributes are declared as REQUIRED in the DTD. Optional attributes can be handled in a manner similar to case 4, while enumerated attributes can be handled similarly to case 6 below.
3. If v is labeled with "," the children of v are processed in depth-first order.
4. If v is labeled with "?" then if the current SAX event matches a descendant of v, the module outputs a 1-bit to the structure stream and processes the subtree rooted at the child of v; otherwise it outputs a 0-bit to the structure stream and skips the descendants of v.
5. If v is labeled with "$*$" or "$+$" then if the current SAX event matches a descendant of v, the module outputs a 1-bit to the structure stream, processes the subtree rooted at the child of v, and then processes v again; if the current SAX event does not match a descendant of v, it outputs a 0-bit to the structure stream and skips the descendants of v.
6. If v is labeled with "$|$" (a choice node) then the current SAX event must match a descendant of one child of v. Assume that the index of this child is i, with the leftmost child having index 0. The module outputs i to the structure stream and processes the subtree rooted at the child of v.

3.3. Example Execution of the DSP Algorithm

We now illustrate further the underlying ideas of the DSP algorithm by using the example XML document of Figure 7, which conforms to the DTD given in Figure 5. When the document is parsed, the stream of SAX events shown in Figure 8 is generated. The DTD tree and the SAX event stream are then processed by the DSP module. In Figure 9 we show the DSP process, which starts from the DTD tree's root node (i.e. the "library" node).

As the first SAX event token, which is a "library" start-element event (i.e. *Token 0* in Figure 8), matches the current DTD tree node (an element node), the module tra-

Token0: Start element – "library"
Token1: Start element – "entry"
Token2: Start element – "author", att0:name="Tom"
Token3: End element – "author"
Token4: Start element – "title"
Token5: PCDATA – "Comp123: Operating Systems – Introduction"
Token6: End element – "title"
Token7: Start element – "year"
Token8: PCDATA – "2003"
Token9: End element – "year"
Token10: Start element – "course_note"
Token11: End element – "course_note"
Token12: Start element – "num_copy"
Token13: PCDATA – "3"
Token12: End element – "num_copy"
Token15: End element – "entry"
Token16: Start element – "entry"
Token17: Start element – "author", att0:name="Jess Chu"
Token18: End element – "author"
Token19: Start element – "title"
Token20: PCDATA – "A Better World"
Token21: End element – "title"
Token22: Start element – "year"
Token23: PCDATA – "1874"
Token24: End element – "year"
Token25: Start element – "publisher"
Token26: PCDATA – "Clear LTD."
Token27: End element – "publisher"
Token28: Start element – "book"
Token29: End element – "book"
Token30: Start element – "num_copy"
Token31: PCDATA – "3"
Token32: End element – "num_copy"
Token33: End element – "entry"
⋮
Token n : End element – "entry"
Token$n+1$: End element – "library"

Fig. 8. A SAX Event Stream Generated Based on the XML Given in Figure 7

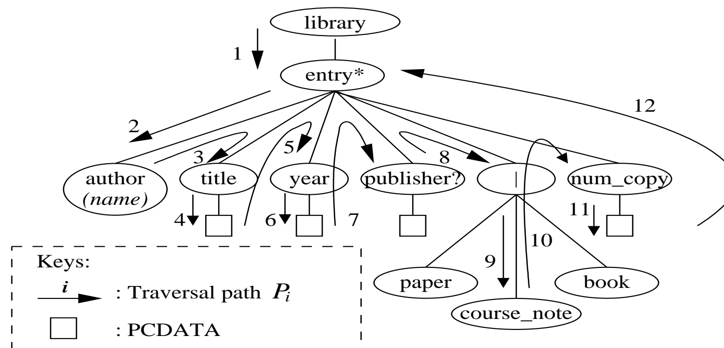


Fig. 9. Parsing of the Library DTD Tree

verses to the subtree of the "library" node in a depth-first manner using path P_1 in Figure 9. Hence, the "entry$*$" node becomes the current DTD tree node. The module then processes the second SAX event token in the SAX event stream.

Since the "entry$*$" node is labeled with the "$*$" repetition operator, a bit is output, the value of which depends on whether the current SAX event token matches the current DTD tree node. As the second SAX event token is an "entry" start-element event (i.e. a match), the module outputs a 1-bit and then traverses the path P_2 as shown in Figure 9.

The module then processes the third SAX event, which is an "author" start-element event (i.e. *Token 2* in Figure 8). Since the occurrence of this element is required by the DTD, nothing is output to the structure stream. However, since "author" possesses a "name" attribute, the attribute value, which comes with the SAX event token, is output to the data stream specified by the path "/library/entry/author/@name". The module then receives an "author" end-element event (i.e. *Token 3* in Figure 8) and traverses to the next child node of the "entry$*$" node (i.e. the "title" node).

Similarly, when the module finds that the next SAX event token, which is a "title" start-element event (i.e. *Token 4* in Figure 8), matches the current DTD tree node, it traverses to its subtree and reaches the PCDATA node using path P_4. The module then expects a PCDATA event. When this event occurs with value "Comp123: Operating Systems - Introduction" (i.e. *Token 5* in Figure 8), the value is output to the data stream whose path is "/library/entry/title/text()". The module then receives a "title" end-element event (i.e. *Token 6* in Figure 8) and traverses to the next child node of the "entry*" node (i.e. the "year" node) using the path P_5.

The subtree under the "year" node is processed in a similar manner to the "title" node. The module then reaches the "publisher?" node and waits for the next SAX event to occur. As the "publisher" node is labelled with an optional operator "?", a bit is output to the structure stream. Since the next incoming SAX event is not a "publisher" start event but rather a "course_note" start event (i.e. *Token 10* in Figure 8), the module outputs a 0-bit. The module then traverses along path P_8 to the next child of the "entry*" node, which is the choice ("|") node.

When the module reaches the choice node, it checks which child of the choice node matches the current SAX event. In this example, the module finds it is the second child, which has an index 1, so it outputs a byte with value 1 to the structure stream. The module then traverses path P_9, followed by path P_{10} when it receives a "course_note" end element event. The "num_copy" node is then processed in a similar manner to the "title" node.

After processing the "num_copy" node, the module returns to the "entry$*$" node using path P_{12} and waits for the next SAX event to occur. Since the next SAX event token is another "entry" start-element event, the process repeats in the manner described above. The PDF traversal continues until all the tokens in the SAX event stream are processed. After the DSP process is finished, a set of output streams that correspond to the structural information and the path-based grouped data values (PCDATA and attribute values) of the input XML document will have been generated.

4. Partitioned Path-Based Grouping

In this section, we discuss the PPG strategy, which is adopted to support *partial decompression* of a compressed document during query evaluation. We impose a minimal indexing scheme over compressed XCQ documents in order to facilitate better query processing. Finally, we present a cost analysis of PPG when evaluating selection and aggregation queries.

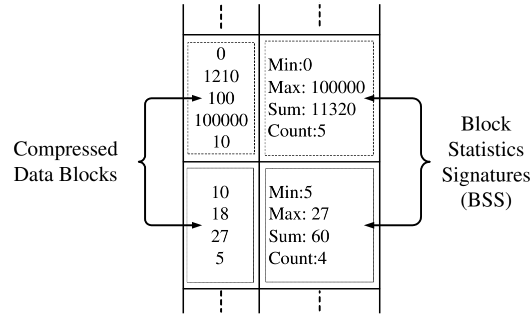


Fig. 10. BSS Indexes in a PPG Data Stream

4.1. PPG Data Streams and BSS Indexing

As we have discussed in Section 3, the DSP module outputs the data elements of an XML document to their corresponding data streams based on their *tree paths* in the *DTD tree*. Within the partitioned path-based data grouping (PPG), XCQ also partitions the data streams into their corresponding sets of blocks, as shown in Figure 1. Each of these data blocks can be compressed or decompressed as an individual unit. This partitioning strategy helps the underlying generic text compressor in XCQ to explore and eliminate redundancies in the input data when these streams are compressed individually, thus increasing the overall compression ratio (Liefke and Suciu, 2000; Sundaresan and Moussa, 2001).

As data elements are stored in a compressed data block in their corresponding data streams, the minimum unit of data access in XCQ is a single compressed data block. Unfortunately, accessing a compressed data block can still be costly. This is because when a data block is accessed, XCQ needs to load the required data block from disk into main memory and then to decompress it, which involves the following three major costs: the disk seek time during block searching, the data transfer time during block fetching, and the processor time used during block decompressing and scanning.

In order to avoid unnecessary access to those data blocks that are irrelevant to the input query, we impose a minimal indexing scheme over a PPG data stream called *Block Statistic Signature* (BSS) indexing. BSS indexing over PPG data streams is minimal in the sense that the scheme requires a very small amount of space and time resources in XCQ. This indexing scheme is a simplified version of the *signature file indexing* approaches (Faloutsos and Christodolakis, 1985; Lin and Faloutsos, 1992; Datta and Thomas, 1999). Like *Projection Signature Indexing* in (Datta and Thomas, 1999), BSS indexing is desirable for indexing block-oriented compressed data. We restrict our discussion of the BSS indexes that are created for those data streams that comprise only numerical data. Assume that there are n blocks. The *Block Statistic Signature* (BSS) index of the i'th block is given by $B_i = \langle s_i, b_i \rangle$ where b_i represents the set of data items in the compressed block and the BSS index value $s_i = \langle min(b_i), max(b_i), sum(b_i), count(b_i) \rangle$, where min, max, sum and $count$ are the usual operations. We define the *value range* for a BSS indexed block B_i, denoted as l_i, by $\langle min(b_i), max(b_i) \rangle$. The same principle can be applied to alphabetical data with some modification of the BSS index values.

Figure 10 depicts the underlying idea of the BSS indexing scheme. When generating a PPG data stream for numerical data values, a *statistical signature* is generated for each compressed data block. The signature summarizes the data values inside the block.

When a query is evaluated, the compressed data blocks in relevant data streams are accessed by XCQ. If BSS indexes are built on the data streams, a *filtering process* is carried out by XCQ as follows. Before a data block is fetched from the disk, XCQ consults the corresponding BSS index and ignores those data blocks that do not contain the required record(s). To do this, XCQ checks the BSS signature of the data block and decides whether the value range of that block overlaps with the value range specified in the query. If the two ranges overlap, which means that the data block *may* contain the required record(s), then the data block is fetched and decompressed for evaluation. If the two ranges do not overlap, the block does *not* contain the required record(s), in which case the data block is not fetched.

4.2. Query Processing in XCQ

In this section we do not intend to present a detailed evaluation of XCQ queries in the scope of this paper, since the mechanism of processing XCQ queries and the optimization issues involved need the full space of another paper. Thus, we now only highlight the principle that PPG data streams help to process some XPath query fragments over a compressed XML document that conforms to the DTD in Figure 5.

We assume that the data streams for the compressed document have been arranged as shown in Figure 11. Let us consider the path query fragment

Q_1 = "entry[author/@name='Jess' and publisher/text()='Clear Ltd.']".

This query fragment selects those entry elements that have both an author with name 'Jess' and a publisher whose value is 'Clear Ltd.'. The evaluation of the query depends on both the document structure and the data values. The evaluation of the two predicates in Q_1 involves data streams d_0 and d_3. As d_0 and d_3 contain string values, neither has a BSS index associated with it. We first explain how XCQ evaluates the first predicate "author/@name ='Jess'", in Q_1. Since there is no BSS index on d_0, XCQ needs to decompress the whole data stream and to test each record in the stream against the value "Jess". Assuming there are two records, r_1 and r_2 in d_0, satisfying the first predicate, XCQ then needs to find the corresponding "publisher" records to evaluate the second predicate[1]. To find the corresponding record indexes, XCQ parses the structure stream against the DTD tree and calculates the record indexes in data streams $d_1, \ldots, d_4$ that correspond to the matched record indexes in d_0. Assume that record s_1 in the first block and record s_2 in the second block of data stream d_3 are the publisher records corresponding to the name records that satisfy the first predicate[2]. XCQ decompresses only the first and second blocks of d_3 and retrieves the two matched records to evaluate the second predicate. Now assume that only record s_1 satisfies the second predicate in Q_1. In order to construct the result, XCQ then decompresses the corresponding blocks in data streams d_1, d_2 and d_4. The blocks needed are calculated from the matching record indexes that were found during structure stream parsing. Assuming that the required records are each in the first block of the corresponding data stream, XCQ needs to decompress only the shaded blocks in Figure 11 when processing Q_1.

We can see that a smaller block size (i.e. using a finer partitioning) helps to improve query performance, since a more precise portion of the compressed document is decompressed during query evaluation. However, there is a trade-off in that finer block parti-

[1] There are fewer elements in stream d_3 than in stream d_0 since they are optional.
[2] XCQ can calculate the blocks corresponding to the record indexes because a fixed blocking factor is used for each data stream.

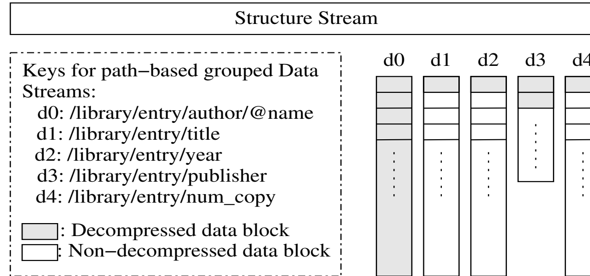


Fig. 11. Decompressed Data Blocks When Processing Q_1

tioning degrades the compression ratio, since fewer redundancies in the data streams can be eliminated by a text compressor. In addition, we see that if the selectivity of the input query increases, the number of data blocks required to be decompressed during the query evaluation will also increase.

The BSS indexes also help to evaluate the query fragments involving aggregation like Q_2 = "count(//entry)" and Q_3 = "sum(//num_copy[text() $>$ 10])". The former counts the number of entry elements in the compressed XML document and the latter sums the values of all those num_copy elements in the compressed XML document that have a value greater than 10. In Q_2, only the number of entry elements in the XML document is needed to generate the answer. Thus, this type of query can be answered without decompressing the data streams. XCQ only needs to parse the structure stream against the DTD tree in Figure 2 once. It then counts and returns the number of "entry$*$" node occurrences that are assigned bit value 1. More complex structural queries can be processed by XCQ using a similar procedure. In Q_3, only one data stream is involved and the result of the query is an aggregate value. In this case, XCQ just needs to find those data values in the data stream d_4 that are greater than 10 and then sum these values. The BSS index constructed for d_4 can be used to filter out those blocks that contain only values less than or equal to 10, allowing XCQ to decompress only a subset of the blocks of d_4 in order to answer the query.

5. XCQ Compression Performance

In this section, we present the experimental results of evaluating the performance of XCQ compression. We study the scalability of XCQ for different sizes of XML documents, and examine critically the impacts of varying PPG block sizes and of imposing BSS indexing on XCQ compression.

5.1. Experimental Design and Setup

We compare the performance of XCQ with that of the following three compressors: (1) *gzip*, which is a widely used generic text compressor, (2) *XMill*, which is a well known XML-conscious compressor, and (3) *XGrind*, which is a well-known XML-conscious compressor that supports querying of compressed XML data.

All the experiments were run on a notebook computer with the following configuration:

– PIII machine with a clock rate of 600MHz.

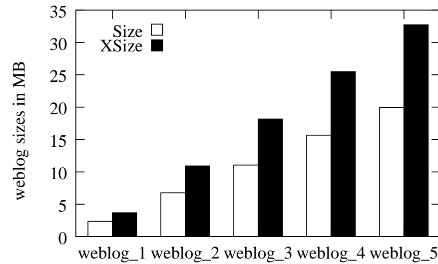


Fig. 12. Original Document Size vs XML-ized Document Size of the Weblog Data Set

– 192 MB RAM of main memory.
– 20 GB hard disk (Ultra DMA/66, 4200 rpm, 512 KB cache, 12 ms seek time).

During the experiments, the number of processes running on the machine was minimized in order to reduce unrelated influences. The time taken to compress and decompress the documents is obtained by running the corresponding processes repeatedly five times and taking the average of the last three runs. The main reason for doing this is to reduce the disk I/O influences on the results by loading the whole document into the physical memory if possible (the same technique is also used in (Liefke and Suciu, 2000)).

To evaluate the performance of the compressors, we used six datasets that are commonly used in XML research (see the experiments in (Cheney, 2002; Liefke and Suciu, 2000; Tolani and Haritsa, 2002)): *Weblog*, *SwissProt*, *DBLP*, *TPC-H*, *XMark*, and *Shakespeare* (Bosak, 1999; Ley, 2005; Apache Software Foundation, 2005; Swiss-Prot, 2005; TPC-H, 2004; XMark, 2003). We now briefly introduce each dataset.

1. *Weblog* is constructed from the Apache webserver log (Apache Software Foundation, 2005). The original documents are not in XML.
2. *Swissprot* is constructed from the documents in the SwissProt database (Swiss-Prot, 2005) with the DNA sequences dropped. The original documents are not in XML.
3. *DBLP* is a collection of the XML documents freely available in the DBLP archive (Ley, 2005). The documents are already in XML format.
4. *TPC-H* is an XML representation of the TPC-H benchmark database, which is available from the Transaction Processing Performance Council (TPC-H, 2004).
5. *XMark* is an XML document that models an auction website. It is generated by the tool provided in (XMark, 2003).
6. *Shakespeare* is a collection of the plays of William Shakespeare in XML (Bosak, 1999).

The first five data sets given above are regarded as *data-centric* as the XML documents have a very regular structure, whereas the last one is regarded as *document-centric* as the XML documents have a less regular structure. It is worth mentioning that the XML-ized weblog dataset (XSize) is about 1.6 times bigger than its non-XML-ized data counterpart (Size), as shown in Figure 12. This is due to the fact that we need to insert control information, such as element tags, into the documents.

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

Table 1. Comparing Compression Ratios CR_1 and CR_2

5.2. Notion of Compression Ratio

There are two different expressions that are commonly used to define the *Compression Ratio* (CR) of a compressed XML document (see the different definitions used in (Cheney, 2002; Liefke and Suciu, 2000; Min et al, 2003; Tolani and Haritsa, 2002)):

$$CR_1 = \frac{sizeof(compressed\,file) \times 8}{sizeof(original\,file)}\ bits/byte.$$
$$CR_2 = (1 - \frac{sizeof(compressed\,file)}{sizeof(original\,file)}) \times 100\%. \quad (1)$$

The first compression ratio, denoted CR_1, expresses the *number of bits required to represent a byte*. Using CR_1 a better performing compressor achieves a relatively *lower* value. On the other hand, the second compression ratio, denoted CR_2, expresses the *fraction of the input document eliminated*. Using CR_2, a better performing compressor achieves a relatively *higher* value.

We now illustrate the difference in both CR definitions by listing compression ratios achieved by gzip and XMill in Table 1. CR_2 shows that the fraction of an input document eliminated by gzip is only a few percent smaller than that of XMill. This means that the performance of gzip and XMill based on CR_2 appear to be similar. However, the actual size of a document compressed by XMill is generally much smaller than that of the the document compressed by gzip. For example, for the Weblog document the size after compression by XMill is about 60% of the size after compression by gzip. This is also true for the SwissProt, TPC-H and XMark documents. On the other hand, as we can see in Table 1, the difference is better reflected by the ratio CR_1. For example, we can see from Table 1 that there is an eleven-fold difference between the CR_1 values for the Weblog (0.177 bits/bytes) and Shakespeare (2.016 bits/bytes) datasets using XMill, while the difference between the corresponding CR_2 readings (i.e. 97.8% and 74.8%) is only 23%. In addition, the notion behind CR_1 (i.e. the number of bits required to represent a byte) gives us an intuition related to the *amount of information* in the dataset, a commonly used notion in information theory (Shannon, 1948). Thus, we henceforth choose to adopt CR_1 as the metric to measure compression performance.

5.3. Compression Performance of XCQ

We now present an empirical study of XCQ performance with respect to compression ratio, compression time, and decompression time. All the numerical data used to construct the graphs can be found in the tables listed in (XCQ Appendix, 2005).

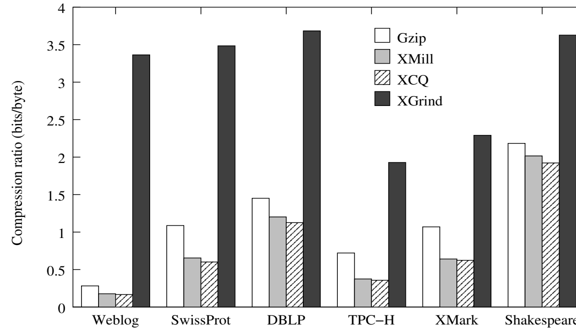


Fig. 13. Compression Ratio for Different Datasets

5.3.1. Compression Ratio

Figure 13 shows the compression ratios that are achieved on the above-mentioned six datasets expressed in CR_1 (bits/byte). Notably, both XMill and XCQ consistently achieve a better compression ratio than gzip. The compression ratio achieved is relatively high for data-centric documents (i.e. Weblog, SwissProt, DBLP, TPC-H and XMark) and relatively low for document-centric documents (i.e. Shakespeare). This can be explained by the fact that the Shakespeare document does not have a regular structure, and therefore XMill and XCQ cannot take much advantage of the document structure during compression.

It is interesting to note that the compression ratio achieved by XGrind is much worse than that achieved by the other three compressors. This is due to the fact that XGrind independently compresses the data values inside an XML document, which is one of the requirements of its *homomorphic transformation* (Tolani and Haritsa, 2002). Thus, XGrind cannot take full advantage of eliminating the redundancies among data values within a document. We now show that, in a statistical sense, XCQ achieves a significantly better compression ratio than XMill. The evidence for this is obtained from performing *formal hypothesis testing for two sample means* (cf. Chapter 8 in (Schefler, 1988)) on a set of 30 different XML datasets as follows.

Let Δ denote the difference in compression ratio between XMill and XCQ on an XML document.

$$\Delta = \text{Compression Ratio of XMill} - \text{Compression Ratio of XCQ}.$$

Let the mean of Δ be denoted as μ. We check the following two hypotheses (the first is the null hypothesis and the second is the alternative hypothesis):

$$\begin{aligned} H_0 &: \mu = 0 \\ H_A &: \mu > 0 \end{aligned} \tag{2}$$

In these hypotheses, H_0 represents the fact that XMill achieves an equally good compression ratio (i.e. there is no statistical difference), and H_A represents the fact that XCQ achieves a better compression, which involves a one-sided test on the positive region of the distribution curve. From the results given in Table 10 in (XCQ Appendix, 2005), we find that the sample *mean* ($\overline{x}$) and *variance* (σ^2) are 0.034 and 0.000622, respectively. It should be pointed out that the Central Limit Theorem (see Chapter 6.4 in (Schefler, 1988)) allows us to assume that the sampling distribution will be approx-

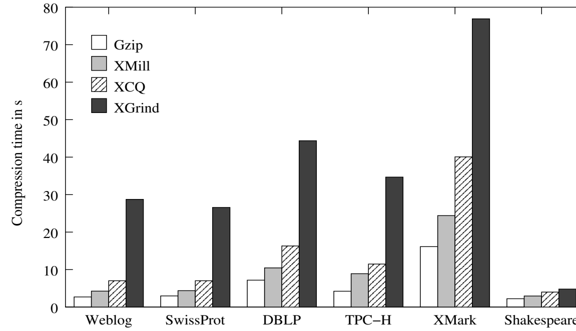


Fig. 14. Compression Time for Different Datasets

imately normal, even though our data may not be distributed normally in the parent populations. We now use the z-test to reject H_0, which is a standard statistical technique. Using the values of $\overline{x}$ and σ^2 above, we compute that $z\text{-}value = 7.476$. If we set the significance level of $\alpha = 0.01$ (note that this is stricter than the acceptable level of $\alpha = 0.05$), the $critical\ value$ = 2.33. As the $z\text{-}value > 2.33$, the null hypothesis H_0 is rejected on the positive side of the distribution curve, which means that H_A is supported. In other words, XCQ achieves a better compression ratio than XMill with a confidence level of 99%. This indicates the effectiveness of our compression approach. With the knowledge of the DTD, XCQ does not need to encode as much structural information as XMill does in the compressed documents.

5.3.2. Compression Time

Figure 14 shows the compression time (expressed in seconds) required by the compressors to compress the XML documents. It is clear that gzip outperforms the other compressors in this experiment. XMill had a slightly longer compression time than gzip, and XCQ in turn had a slightly longer compression time than XMill. The time overhead can be explained by the fact that both XMill and XCQ introduce a pre-compression phase for re-structuring the XML documents to help the main compression process. In the pre-compression phase, XCQ generates precise PPG data streams by recursively traversing the DTD tree. In contrast, XMill adopts by default an *approximation match* on a *reversed DataGuide* to determine which container a data value belongs to. This *grouping by enclosing tag* heuristic runs faster than the grouping method used in XCQ and thus XMill runs slightly faster than XCQ. It should be noted, however, that the data grouping result generated by XMill may not be as precise as our PPG data streams. This complicates the search for related data values of an XML fragment in the separated data containers in a compressed file. In addition, the compression buffer window size in XMill is set at 8MB, which is optimized solely for better compression (Liefke and Suciu, 2000). Such a large chunk of compressed data is costly in full or partial decompression. On the other hand, the compression time required by XGrind is generally much longer than that required by gzip, XMill, and XCQ. XGrind uses Huffman coding and thus needs an extra parse of the input XML document to collect statistics for a better compression ratio, resulting in almost double the compression time required in a generic compressor (Tolani and Haritsa, 2002).

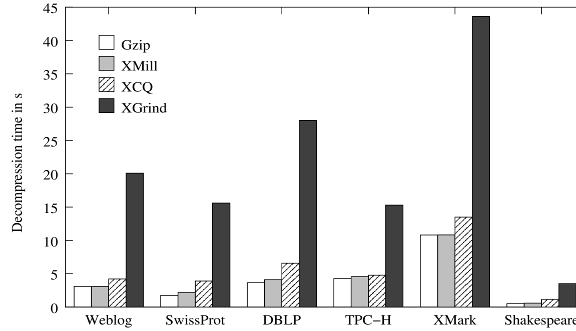


Fig. 15. Decompression Time for Different Datasets

5.3.3. Decompression Time

Figure 15 shows the decompression time (expressed in seconds) required by the decompressors. One observation from Figure 15 is that, in general, gzip outperforms the other compressors in decompression and XMill runs faster than XCQ. Another observation is that XGrind requires a much longer decompression time than the other five decompressors. We also note that XMill decompresses Weblog documents slightly faster than gzip, which conforms to the results reported in (Liefke and Suciu, 2000).

The extra overhead required by XMill and XCQ to merge data items into their original positions in the structure after decompressing the data containers (or data streams) may explain longer decompression times compared to gzip. However, when the XMill-compressed file size is much smaller than the gzip-compressed file size, as shown in the case of the XMark dataset, it is possible that XMill achieves a decompression time that is shorter than that of gzip, mainly due to the much smaller disk read overhead.

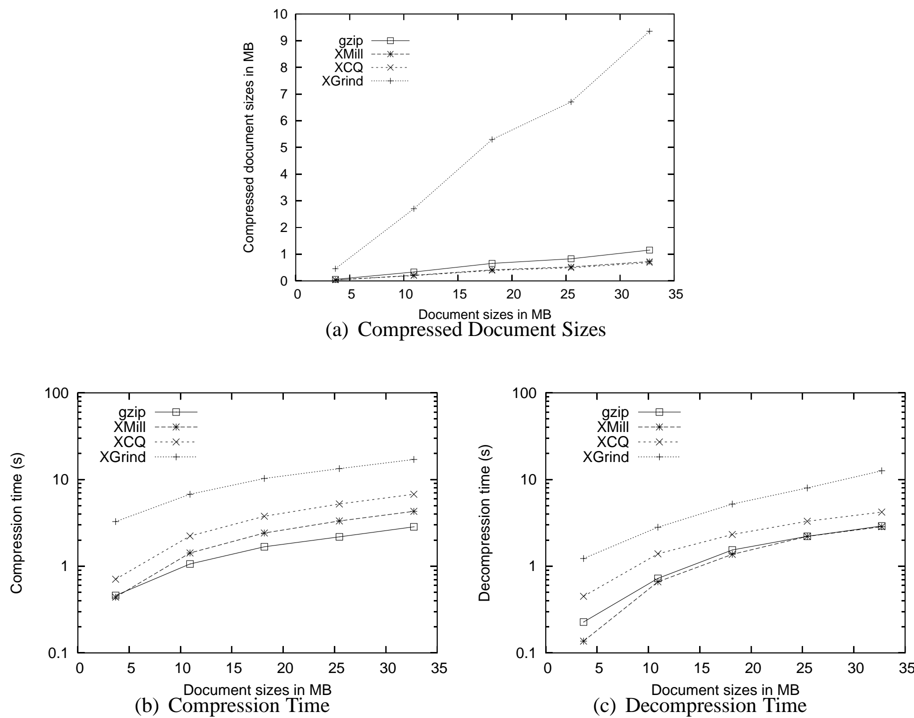
5.3.4. Scalability of XCQ Compression

We now study the scalability of XCQ with respect to the other compressors. As we have observed that the compressors behave in a similar way for different document types, we choose to use *Weblog* documents of different sizes, presented in Figure 12, as the data set in this experiment.

Figure 16(a) shows the comparison between compressed document sizes (expressed in MB) obtained by different compressors. All four compressors scale roughly linearly with respect to the input document size, which is consistent with the findings shown in Figure 13. XCQ and XMill produce compressed documents of very similar sizes, while the poor performance of XGrind (consistently large gradient) is expected according to Figure 13.

Figure 16(b) shows the performance of the compressors in terms of compression time (expressed in seconds), presented on a logarithmic scale. Clearly, gzip outperforms the other compressors consistently regardless of the document size. In particular, both XMill and XCQ have a longer compression time than gzip for all documents, since they introduce a pre-compression phase. XMill takes about 1.6 times longer than gzip to complete the compression process, a finding consistent with the results given in (Liefke and Suciu, 2000), while XCQ, in turn, takes about 1.6 times longer than XMill. XGrind takes considerably longer than XCQ.

Figure 16(c) shows the performance in terms of decompression time (expressed in seconds), presented on a logarithmic scale. It can be seen that XMill completes the de-



(a) Compressed Document Sizes

(b) Compression Time

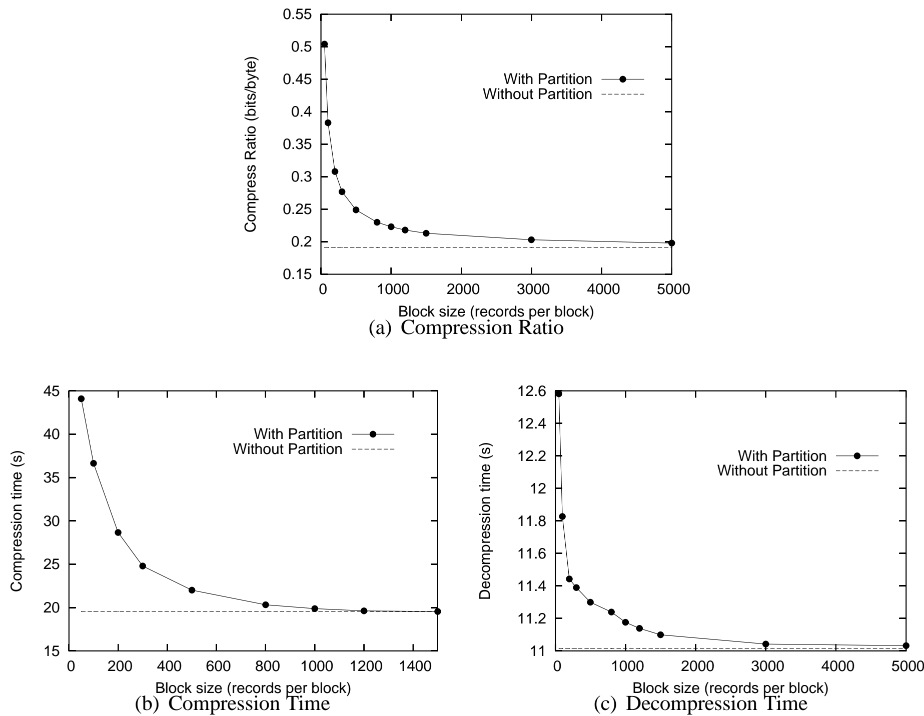
(c) Decompression Time

Fig. 16. Processing Weblog XML Documents in XCQ

compression process either more quickly than or in roughly the same time as gzip. This is consistent with the results in (Liefke and Suciu, 2000). However, on the other benchmark XML documents we used, such as DBLP, Shakespeare, SwissProt and TPC-H, XMill requires a slightly longer decompression time than gzip (cf. Figure 14). XCQ takes 1.3 seconds longer than XMill to decompress a 32MB Weblog XML document. However, it should be noted that XCQ is able to process queries by only partially decompressing the document, implying that the decompression overhead will be much lower.

5.3.5. Summary and Discussion

To summarize, we find that both XMill and XCQ achieve better compression ratios than gzip at the expense of compression and decompression time. XCQ needs more time than XMill to generate a PPG data stream in an XML document when the document is compressed. This enables XCQ to achieve a slightly better compression ratio than XMill. On the other hand, the compression performance of XGrind is consistently worse than those of XMill and XCQ. This supports the findings reported in (Tolani and Haritsa, 2002).



(a) Compression Ratio

(b) Compression Time

(c) Decompression Time

Fig. 17. Effect of Block Partitioning

5.4. Block Partitioning and BSS Indexing

In this section, we study the impact of varying the block size and imposing BSS indexing on data streams in XCQ. We only present the effect on the 89MB Weblog dataset, since other datasets exhibit similar behavior.

5.4.1. Effect of Block Partitioning

We now present the results related to the choice of block size when partitioning PPG data streams.

Figure 17(a) depicts the compression ratio that is achieved by XCQ under different block sizes. For ease of reference, we superimpose a dotted line on the figure to indicate the compression ratio achieved by XCQ when no partitioning is made. It can be seen from the figure that the compression ratio degrades when a smaller block size (i.e. a finer partitioning) is used. The degradation in the compression ratio is due to the fact that fewer redundancies in the data streams can be eliminated by a text compressor if each block is compressed as a finer individual unit. When the block size is increased to around 5,000 records per block, the compression ratio achieved is comparable to that achieved when no partitioning is made; the difference is less than 4%. Figures 17(b) and 17(c) show that both the compression and decompression times are also degraded when a finer partitioning is used. The degradation in the compression and decompression times is due to the fact that, if we set a smaller block size in XCQ, the number of compression and

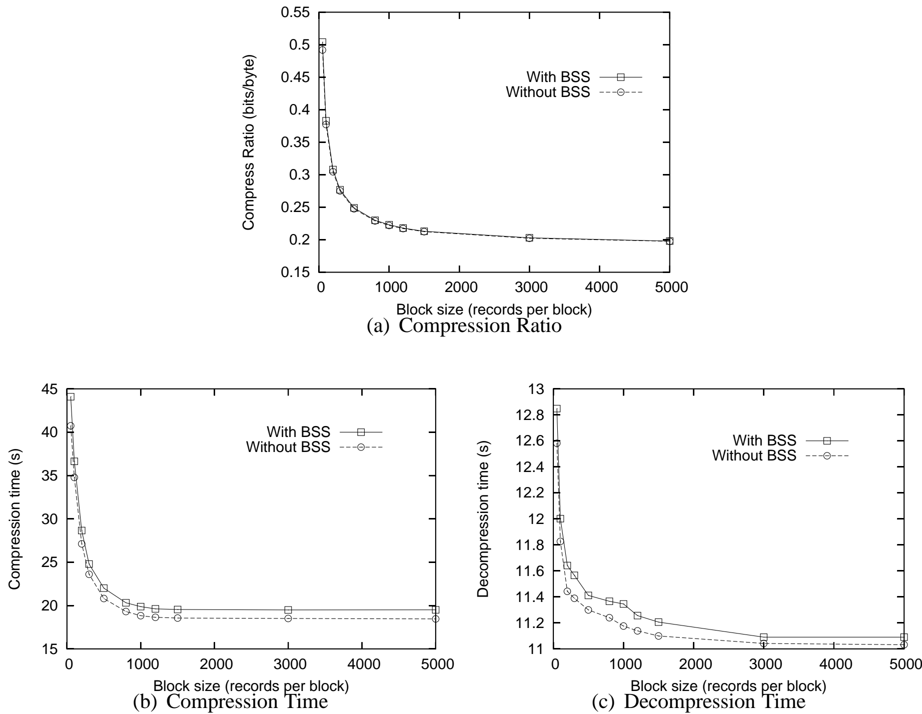


Fig. 18. Effect of BSS Indexing

decompression operations is increased. Consequently, the total overhead is increased, since each block is compressed and packed as an individual unit.

5.4.2. Effect of BSS Indexing

We now study the impact of BSS indexing on XCQ compression. In the following, we compare the performance of two configurations of XCQ: the first is the default configuration with BSS indexing, while the second is XCQ with BSS indexing *turned off*.

Figures 18(a) to 18(c) show comparisons of the compression ratio, compression time, and decompression time, respectively, between the two configurations of XCQ. As can been seen, only small overheads are added to the compression (roughly 5%) and decompression (roughly 1% to 2%) times when BSS indexing is adopted in XCQ. There is virtually no difference in the compression ratio between the two XCQ versions (i.e. with and without BSS indexing). These results agree with our expectations, since the BSS indexing scheme is designed to be minimal for block-oriented compressed data.

5.4.3. Discussion

We find that a very fine partitioning (smaller than 3000 records per block) on PPG data streams imposes overheads in compression ratio, compression time, and decompression time. However, when using a finer data stream partitioning, XCQ can utilize the advantage of decompressing a more precise portion of the compressed document when

answering queries, as was discussed in Section 4.2. We need small blocks in a PPG data stream in order to have efficient XCQ query processing. On the other hand the overhead of BSS indexing is minimal, and is virtually independent on the block size in our study.

6. Related work

Because there are usually substantial redundancies embedded in an XML document structure, information theory states that we should be able to achieve significant compression of XML data. However, such embedded redundancies are not trivial to discover and are largely ignored by conventional textual compression such as gzip (Gailly and Adler, 2003a) or bzip2 (Seward, 2005). Thus, many XML-conscious compression technologies have been proposed and developed in recent years.

We are aware of two XML-conscious compression technologies that make use of DTDs. They are *Differential DTD Tree* (DDT) compression in Millau (Sundaresan and Moussa, 2001) and the Structure Compression Algorithm (SCA) proposed in (Levene and Wood, 2002). The DDT and SCA approaches adopt a similar compression strategy that encodes only the information that cannot be inferred from a given DTD. (A similar approach to encoding a document with respect to a DTD was used for a different purpose in (Garofalakis et al, 2003).) The limitation of these approaches is that, when parsing an XML document in order to create a corresponding tree structure, a large amount of memory is required to store the generated DOM tree. The vigorous use of virtual memory leads, in practice, to frequent thrashing of disk I/O, which degrades the efficiency of the compression process.

XMill (Liefke and Suciu, 2000) is a typical example of unqueriable XML compression technology[3]. It achieves a good compression ratio but the compressed data needs to go through a full decompression in order to evaluate queries. XMill has a pre-compression phase introduced prior to the main compression process. The pre-compression phase is designed to perform the following two main tasks: first, to separate the document structural information from the data, and second, to group data items with related semantics in the same "container". The structural information includes element tag names and attribute names. The data items include PCDATA and attribute values.

In order to group data items in an effective manner, XMill uses an approximation matching on the *reversed DataGuide* (Goldman and Widom, 1997; Liefke and Suciu, 2000) to determine which containers data values belong to. In its default setting, data items with the same tag or attribute name are grouped in the same data container. Each container is then compressed individually in the main compression phase by using an ordinary text compressor such as gzip, whose output is then concatenated as a single file. In addition, path expressions can be specified as command line arguments to instruct the XMill compressor how to group data items. Specific semantic compressors can also be employed in order to pre-compress the corresponding data containers before they are compressed by a text compressor. This further helps to achieve a better compression ratio. However, user expertise and manual effort are needed to intervene in the compression process.

Cheney describes a different XML-conscious encoding called *Multiplexed Hierarchical Modeling* (MHM) in (Cheney, 2002). This offers better compression ratios than XMill at the expense of compression speed. Like XMill, the compressed documents need to be decompressed before queries can be evaluated on them.

[3] To the best of our knowledge, only XMill source code is released and directly compilable.

To avoid the need to decompress documents when evaluating queries, some recent XML compression technologies provide direct access to compressed documents. XGrind (Tolani and Haritsa, 2002) is the first known *queriable* XML compressor. XGrind adopts a homomorphic transformation strategy to transform an XML document into a specialized compressed format that preserves the syntactic and semantics information of the original document. All the tag and attribute names in the compressed document are tokenized using a dictionary encoding approach, and enumeration-type attribute values are binary encoded. PCDATA and general attribute values are compressed individually by using non-adaptive context-free Huffman encoding (Huffman, 1952).

As the compressed document output by XGrind is a homomorphic transformation of the input document, all operations that can be executed over the original document, such as querying, are preserved. These operations can be executed using existing techniques and tools with some modifications. However, it should be noted that the advantage of avoiding decompression[4] when querying is obtained at the expense of compression ratio. For instance, XGrind compresses an 89MB Weblog XML document into a 38MB compressed document, while XMill is able to compress the same document to only 2.3MB.

Recent work by Buneman et al. (Buneman et al, 2003; Buneman et al, 2005) and on XPRESS (Min et al, 2003) also allow queries to be evaluated directly on compressed XML documents. The technique adopted in (Buneman et al, 2003) compresses the *skeleton* of a given XML document (essentially its structure) by using a technique based on the idea of sharing common subtrees, thereby transforming the skeleton into a directed acyclic graph (DAG). This DAG can be further compressed by replacing any consecutive sequence of out-edges to the same vertex by a single edge labeled with the appropriate cardinality. The focus of Buneman et al.'s skeleton framework is different from our approach, in that skeleton compression aims at reducing the size of the document structure, rather than the textual data items in the document, and the framework does not use knowledge of a DTD to perform structure compression. In (Buneman et al, 2005), the skeleton and its corresponding data storage (or data vectors) are used together to support processing a fragment of XQuery. We can view the skeleton component as a "pseudo-DTD", since, roughly speaking, it presents a compact structure of a given XML document, while a data vector in (Buneman et al, 2005) is essentially a non-partitioned data stream.

Although the skeleton technique is able to compress the structure of an XML document well, the overall compression ratio (including textual data) achieved by this framework, as mentioned in (Buneman et al, 2003), is worse than that of XMill. However, using the proposed compression technique, the authors formally study the evaluation of expressions in Core XPath (Buneman et al, 2003) and XQuery (Buneman et al, 2005). The essence of evaluating such queries is done by manipulating the compressed skeleton instance with only partial decompression. This technique allows the navigational aspect of query evaluation, which is responsible for a large portion of the query processing time, to be carried out in main memory. Such techniques are complementary to our work. In principle, we could extend our work to output a query result in a compressed format (i.e. outputting the related fragment of the structure stream and the data according to their corresponding data streams).

XPRESS adopts mixed encoding methods on paths and achieves a much better query processing time (two to three times faster) than that of XGrind, according to the experimental results reported in (Min et al, 2003). However, as also mentioned in (Min et

[4] Note that range queries still require partial decompression in XGrind.

al, 2003), the compression ratio of XPRESS is in fact worse than that of XMill. In addition, compression time is almost twice that of XMill, since XPRESS requires parsing the input XML document twice in the compression process.

XQuec (Arion et al, 2004), which is a recent emerging XML compression technology, claims that XQuery language can be fully supported. Like XGrind and XPRESS, XQueC is able to compress an XML document as well as to avoid full decompression during query evaluation. However, the approach differs from that used by XGrind and XPRESS in that XQueC separates the XML structure from the XML data items and uses a variety of auxiliary structures, such as DataGuides (Goldman and Widom, 1997), structure trees, and other indexes, in order to support efficient evaluation of XQuery (Boag et al, 2005). The individually compressed data items are organized into containers, and they can be efficiently accessed by pointers from the auxiliary data structures. However, it seems that the fine-grained compression is very likely to result in a worse compression ratio than that of XMill. Moreover, the auxiliary data structures, together with the pointers to the individually compressed data items, would incur a huge space overhead.

7. Conclusions and Future Work

We have presented the development of XCQ, a prototype system designed to support querying over compressed XML documents. Overall, we showed that by exploiting the information present in a DTD, XCQ is able to achieve better compression and to support evaluation of a set of fundamental XPath queries. Our development is based on the following series of novel techniques.

- We proposed *DTD Tree and SAX Event Stream Parsing* (DSP), which enables users to compress XML documents that conform to a given DTD. DSP does not require user expertise, such as providing data grouping commands, in the compression process.
- We proposed the Partitioned Path-Based Data Grouping (PPG) of data streams as an effective block-oriented storage scheme for supporting partial decompression over compressed data.
- We proposed a simple and minimal indexing scheme for PPG data streams called the *Block Statistical Signature* (BSS) indexing scheme. The BSS indexing scheme is designed to facilitate the fast recognition of target blocks in a PPG data stream.

We demonstrated in a diversified set of experimental results in Section 5 that XCQ can achieve good compression and can compress XML-ized documents consistently better than the generic text compressor gzip. It also achieves a slightly better compression ratio, at the expense of a greater compression time, than state-of-the-art systems such as XMill, which is optimized only for compression ratio. Based on the study conducted in Section 5.3, we found that XCQ achieves a better compression ratio than gzip at the expense of the compression and decompression times. Comparing it to another well-known unqueriable compressor, XMill, XCQ performs slightly better in terms of the compression ratio but worse in compression and decompression time. XCQ was found to be scalable for a wide range of XML benchmark documents, as listed in Section 5.1. We also found that XCQ performs consistently better than another well-known queriable XML compressor, XGrind, in compression performance. Admittedly, the main drawback of DSP is that the compression and decompression times for processing an XML document as a whole are relatively longer than those of the generic compressor gzip. The underlying reason for this is that XCQ needs to generate PPG data

streams corresponding to the compressed XML document. However, we argue that the time overhead is worthwhile, since the generated PPG data streams are able to support queries over compressed documents in an efficient manner. In practice, this time overhead is a once-off consumption, since the generated data streams can be buffered when XCQ is used in the context of an XML application.

The techniques presented in XCQ pave the way to develop a fully-fledged querying engine that is able to support more sophisticated XPath and XQuery queries. Currently, we are also developing a cost model that is able to account for how the response time is affected by various parameters involved in the compression strategy, such as the block size in a data stream, the number of clusters, the cost of scanning indexes, the cost of decompressing a block, query selectivity, data distribution and workload statistics. The cost model can be incorporated into the XCQ engine for optimizing query evaluation. An orthogonal but promising direction related to query optimization is to employ a caching technique in the engine to handle the PPG data blocks of a compressed document. An efficient caching scheme for fetching and updating compressed data blocks would help XCQ to minimize overheads, such as I/O costs, during the compression, querying and updating of XML documents.

In the existing XCQ version, we have not considered the problem of updating compressed XML documents. However, an append operation could be supported by XCQ in a straightforward manner. In order to append an XML fragment to the compressed document, XCQ would first extract the structural information and data information from the fragment and then append the extracted information to the structure stream and the corresponding data streams. Using this approach, only the structure stream and the last block of each updated data stream would have to be re-compressed. However, for general update operations, we still need to devise efficient techniques to deal with the deletion and modification of fragments of compressed XML data.

Although our current implementation supports only non-recursive DTDs, it would be straightforward to modify it to handle recursive DTDs. In the first place, the DTD parser would build a DTD graph rather than a tree. When parsing a document against the DTD graph, the graph would still be traversed in depth-first order. However there would now be the possibility that a node in the graph could be visited multiple times while parsing a single path in the document, although the number of times would be bounded by the maximum depth of any node in the document. In order to determine the correct data stream on which to output a text node, the system could consult the stack maintained by the depth-first search of the DTD graph. This technique effectively implements a deterministic PDA as shown to be necessary and sufficient for the single-pass validation of XML documents by Segoufin and Vianu in (Segoufin and Vianu, 2002).

Acknowledgements. We would like to express our sincere thanks to the editor and the reviewers, who provided very insightful and encouraging comments. This work is supported in part by grants from the Research Grant Council of Hong Kong, Grant Nos HKUST6185/02E, HKSUT6165/03E and DAG04/05.EG10.

References

Apache Software Foundation (2005). Log Files - Apache HTTP Server, `http://httpd.apache.org/docs/logs.html`

Arion A, Bonifati A, Costa G, D'Aguanno S, Manolescu I, and Pugliese A (2004) Efficient Query Evaluation over Compressed XML Data. In Bertino E, Christodoulakis S, Plexousakis D, Christophides V, Koubarakis M, Böhm K, Ferrari E (eds). Proceedings of Advances in Database Technology (EDBT 2004), 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March, 2004. Lecture Notes in Computer Science 2992, Springer, Berlin, pp 200–218

Bell TC, Cleary JG, and Witten IH (1990) Text Compression. Prentice Hall, Englewood Cliffs, New Jersey, USA

Boag S, Chamberlin D, Fernández MF, Florescu D, Robie J and Siméon J (eds) (2005) XQuery 1.0: An XML Query Language. W3C Working Draft, 15 September 2005, `http://www.w3.org/TR/xquery`

Bosak J (1999) Shakespeare 2.00. `http://www.cs.wisc.edu/niagara/data/shakes/shaksper.htm`

Bray T, Paoli J, Sperberg-McQueen CM, Maler E and Yergeau F (eds) (2004) Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation, 4 February 2004, `http://www.w3.org/TR/REC-xml`

Buneman P, Grohe M and Koch C (2003) Path Queries on Compressed XML. In Freytag JC, Lockemann PC, Abiteboul S, Carey MJ, Selinger PG, Heuer A (eds) Proceedings of the 29th International Conference on Very Large Data Bases, Berlin, Germany, September, 2003, pp 141–152

Buneman P, Choi B, Fan W, Hutchison R, Mann R and Viglas S (2005) Vectorizing and Querying Large XML Repositories. Proceedings of the 21th International Conference on Data Engineering, Tokyo, Japan, April, 2005, pp 261–272

Burrows M and Wheeler DJ (1994) A Block-Sorting Lossless Data Compression Algorithm. Technical Report 124, SRC, Digital Equipment Corporation, Palo Alto, California

Cannataro M, Comito C and Pugliese A (2002) SqueezeX: Synthesis and Compression of XML Data. Proceedings of the IEEE International Conference on Information Technology: Coding and Computing, Las Vegas, USA, April 2002, pp 326–331

Cheney J (2001) Compressing XML with Multiplexed Hierarchical PPM Models. Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, March, 2001, pp 163–172

Clarke J (2004) The Expat XML Parser. `http://expat.sourceforge.net/`

Clark J and DeRose S (eds) (1999) XML Path Language (XPath) Version 1.0. W3C Recommendation, 16 November 1999 `http://www.w3.org/TR/xpath`

Cleary J, Teahan W and Witten I (1995) Unbounded Length Contexts for PPM. In Storer JA, Cohn M (eds). Proceedings of the IEEE Data Compression Conference, Snowbird, UT, USA, March, 1995, pp 52–61

Datta A and Thomas H (1999) Accessing Data in Block-Compressed Data Warehouses. Proceedings of the Ninth Workshop on Information Technologies and Systems (WITS), Charlotte, North Carolina, USA, December, 1999

DTDParser - A Java DTD Parser (2005). `http://www.wutka.com/dtdparser.html`

Faloutsos C and Christodoulakis S (1985) Design of a Signature File Method that Accounts for Non-uniform Occurrence and Query Frequencies. In Pirotte A, Vassiliou Y (eds). Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, Sweden, August, 1985, pp 165–170

Gailly J-L and Adler M (2003a) gzip 1.2.4. `http://www.gzip.org/`

Gailly J-L and Adler M (2003a) zlib 1.1.4. `http://www.gzip.org/zlib/`

Garofalakis M, Gionis A, Rastogi R, Seshadri S and Shim K (2003) XTRACT: Learning Document Type Descriptors from XML Document Collections. Data Mining and Knowledge Discovery 7:23–56

Girardot M and Sundaresan N (2000a) Millau: An Encoding Format for Efficient Representation and Exchange of XML over the Web. Proceedings of the 9th International World Wide Web Conference, Amsterdam, The Netherlands, May, 2000, pp 747–765

Girardot M and Sundaresan N (2000b) Efficient Representation and Streaming of XML Content over the Internet Medium. Proceedings of the IEEE International Conference on Multimedia and Expo (I), New York, NY, USA, July/August 2000, pp 67–70

Goldman R and Widom J (1997) DataGuides: Enabling Query Formation and Optimization in Semistructured Databases. In Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds). Proceedings of the 23rd International Conference on Very Large Data Bases, Athens, Greece, August, 1997, pp 436–445

Huffman DA (1952) A Method for Construction of Minimum-Redundancy Codes. Proceedings of the IRE 40:1098–1101

Ishikawa H, Yokoyama S, Isshiki S and Ohta M (2001) Project Xanadu: XML- and Active-Database-Unified Approach to Distributed E-Commerce. In Tjoa AM and Wagner R (eds). Proceedings of the 12th International Workshop on Database and Expert Systems Applications, Munich, Germany, September, 2001, pp 833–837

Iyer B and Wilhite D (1994) Data Compression Support in Databases. In Bocca JB, Jarke M, Zaniolo C (eds). Proceedings of the 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile, September, 1994, pp 695–704

Java Technology (2005). `http://java.sun.com/`

Lam WY, Ng W, Wood PT and Levene M (2003) XCQ: XML Compression and Querying System. Poster Proceedings of the Twelfth International World Wide Web Conference, Budapest, Hungary, May, 2003

Levene M and Wood PT (2002) XML Structure Compression. Proceedings of the Second International Workshop on Web Dynamics, Honolulu, Hawaii, May 2002

Ley M (2005) DBLP. http://dblp.uni-trier.de/

Liefke H and Suciu D (2000) XMill: An efficient compressor for XML Data. In Chen W, Naughton JF, Bernstein PA (eds). Proceedings of the ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA, May, 2000, pp 153–164

Lin Z and Faloutsos C (1992) Frame-Sliced Signature Files. IEEE Transactions on Knowledge and Data Engineering 4(3):281–289

Martin B and Jano B (1999) WAP Binary XML Content Format. W3C NOTE, 24 June 1999, http://www.w3.org/TR/wbxml/

Megginson D (2004) SAX. http://www.saxproject.org/

Min JK, Park MJ and Chung CW (2003). XPRESS: A Queriable Compression for XML Data. In Halevy AY, Ives ZG, Doan A (eds). Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June, 2003, pp 122–133

Ng WK and Ravishankar C (1997) Block-Oriented Compression Techniques for Large Statistical Databases. IEEE Transactions on Knowledge and Data Engineering 9(2):314–328

Poess M and Potapov D (2003) Data Compression in Oracle. In Freytag JC, Lockemann PC, Abiteboul S, Carey MJ, Selinger PG, Heuer A (eds). Proceedings of the 29th International Conference on Very Large Data Bases, Berlin, Germany, September, 2003, pp 937–947

Schefler WC (1988) Statistics: Concepts and Applications. The Benjamin-Cummings Publishing Co., Inc., Redwood City, California, USA

Segoufin L and Vianu V (2002) Validating Streaming XML Documents. In Popa L (ed). Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Madison, Wisconsin, USA, June, 2002, pp 53–64

Seward J (2005) bzip2 and libbzip2. http://www.bzip.org/

Shannon CE (1948) A Mathematical Theory of Communication. The Bell System Technical Journal 27:379–423, 623–656

Sundaresan N and Moussa R (2001) Algorithms and Programming Models for Efficient Representation of XML for Internet Applications. Proceedings of the Tenth International World Wide Web Conference, Hong Kong, China, May, 2001, pp 366–375

Swiss-Prot Protein Knowledgebase (2005). http://www.expasy.ch/sprot/

TAR (2004). http://www.gnu.org/software/tar/

Tolani PM and Haritsa JR (2002) XGRIND: A Query-friendly XML Compressor. Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, February/March, 2002, pp 225-234

Transaction Processing Performance Council (2004) TPC-H: An ad-hoc, decision support benchmark. http://www.tpc.org/tpch/default.asp

XMark – An XML Benchmark Project (2003). http://monetdb.cwi.nl/xml/

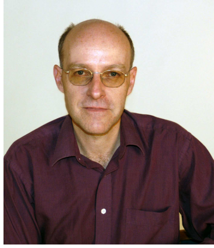
XML Solutions (2000) XMLZIP. http://www.xmls.com/

XCQ Appendix (2005) Experimental Data of XCQ Performance, http://www.cs.ust.hk/∼wilfred/XCQ/appendix.pdf

Author Biographies

Wilfred Ng obtained his M.Sc.(Distinction) and Ph.D. degrees from the University of London. His research interests are in the areas of databases and information Systems, which include XML data, database query languages, web data management, and data mining. He is now an assistant professor in department of computer science, the Hong Kong University of Science and Technology (HKUST). Further Information can be found at the following URL: http://www.cs.ust.hk/faculty/wilfred/index.html.

Wai-Yeung Lam obtained his M.Phil. degree from the Hong Kong University of Science and Technology (HKUST) in 2003. His research thesis was based on the project “XCQ: A Framework for Querying Compressed XML Data.” He is currently working in industry.



Peter Wood received his PhD in Computer Science from the University of Toronto in 1989. He had previously studied at the University of Cape Town, South Africa, obtaining a BSc degree in 1977 and an MSc degree in Computer Science in 1982. Currently he is a Senior Lecturer at Birkbeck and a member of the Information Management and Web Technologies research group. His research interests include database and XML query languages, query optimisation, active and deductive rule languages, and graph algorithms.

Mark Levene received his PhD in Computer Science in 1990 from Birkbeck College, University of London, having previously been awarded a BSc in Computer Science from Auckland University, New Zealand in 1982. He is currently Professor of Computer Science at Birkbeck College, where he is a member of the Information Management and Web Technologies research group. His main research interests are Web search and navigation, Web data mining and stochastic models for the evolution of the Web. He has published extensively in the areas of database theory and web technologies, and has recently published a book called An Introduction to Search Engines and Web Navigation.

Correspondence and offprint requests to: Wilfred Ng, Department of Computer Science, Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. Email: wilfred@cs.ust.hk