XPACK: A HIGH-PERFORMANCE WEB DOCUMENT ENCODING

Daniel Rocco
Department of Computer Science, University of West Georgia
Carrollton, GA 30118 USA

James Caverlee, Ling Liu
College of Computing, Georgia Institute of Technology
Atlanta, GA 30332, USA

Keywords: XML compression, path queries.

Abstract: XML is an increasingly popular data storage and exchange format whose popularity can be attributed to its self-describing syntax, acceptance as a data transmission and archival standard, strong internationalization support, and a plethora of supporting tools and technologies. However, XML's verbose, repetitive, text-oriented document specification syntax is a liability for many emerging applications such as mobile computing and distributed document dissemination. This paper presents XPack, an efficient XML document compression system that exploits information inherent in the document structure to enhance compression quality. Additionally, the utilization of XML structure features in XPack's design should provide valuable support for structure-aware queries over compressed documents. Taken together, the techniques employed in the XPack compression scheme provide a foundation for efficiently storing, transmitting, and operating over Web documents. Initial experimental results demonstrate that XPack can reduce the storage requirements for Web documents by up to 20% over previous XML compression techniques. More significantly, XPack can simultaneously support operations over the documents, providing up to two orders of magnitude performance improvement for certain document operations when compared to equivalent operations on unencoded XML documents.

1 INTRODUCTION

The XML document format (Bray, 1998) is emerging as a popular document encoding for online information exchange. Standardized Web document formats like XML are advantageous for a variety of reasons. XML has well-defined semantics, strong internationalization support (Savourel, 2001), and a plethora of developer tools for managing and exchanging data. In addition, XML derived languages, such as WSDL (Christensen et al., 2001) and SOAP (Mitra, 2003), provide higher level interaction standards that leverage existing XML technology. XML data is self-describing and authors are encouraged to use clear entity names to assist other users in understanding the data (Bray, 1998). Since many parties interested in data exchange interact with different entities during the course of a transaction, predefined data exchange standards are a must. In the highly dynamic world of the Web, the set of data exchange partners an entity may use will evolve over time, which provides a strong argument for the use of industry-standard communication technologies rather than ad hoc solutions.

However, XML has two disadvantages that present obstacles to widespread adoption as an information exchange medium for many applications: the size penalty and textual representation. Many entities that might consider XML would need to convert existing proprietary document formats into XML, which typically produces an undesirable and dramatic increase in the size of the stored data (Liefke and Suciu, 2000). Another concern stems from the fact that XML is stored in a text document encoding like ASCII, which incurs significant computational costs for parsing and validation.

Many applications exist that could benefit from the standardization of XML but require a more efficient document representation. For example, data distribution and routing applications require large numbers of documents to be handled quickly (Altinel and Franklin, 2000), while today's mobile devices have limited processing power, communication bandwidth, and storage capacity. For these and other applications, it is advantageous to minimize the storage space required by documents and to provide efficient access to application-specific areas of interest. While XML provides advantages with its self-describing characteristics and universally recognized format, these applications cannot afford the performance penalty that has previously been the price of converting to XML.

There have been several efforts to reduce the storage impact of converting to or otherwise utilizing XML. XMill (Liefke and Suciu, 2000) is an XML document compressor designed to alleviate the concern of data expansion due to XML conversion. The WAP Binary XML Format (World Wide Web Consortium, 1999) and its extension Millau (Girardot and Sundaresan, 2000) have been introduced for the efficient encoding and streaming of XML structure, particularly in a wireless environment. These compression schemes addressed the data expansion concern but do not provide explicit support for querying compressed documents. There have been subsequent efforts to provide query support over compressed XML documents with systems that compress the content of the document (Min et al., 2003) or its structure (Buneman et al., 2003).

This paper presents XPack, a document compression system providing both good compression and strong query support for compressed documents. The design goal of XPack is to support the acceptance of XML as a viable data exchange mechanism by minimizing the performance penalty incurred by applications that use it. XPack's compression and query techniques are built upon these fundamental design principles:

1. *Redundancy Elimination.* XPack reduces much of the redundancy found in XML documents, yielding a smaller document footprint.
2. *Binary Format.* XPack's binary encoding requires no parsing when loading a document from disk. Document parsing and verification is a resource intensive operation, so applications using XPack can expect better performance when loading documents.
3. *Compartmentalization.* XPack separates various document components to provide faster access to interesting aspects of a document, such as its tree structure information or node tag list.
4. *Compressed Data Access.* Unlike other widely used document compression systems, XPack provides general query facilities that operate over the compressed documents. Compressed data access allows applications to store data in a compact, space-saving format without sacrificing the ability to do ad hoc querying.

2 RELATED WORK

The XPack document encoding leverages ideas from our previous work on the Page Digest system (Rocco et al., 2003) for efficient representation of HTML Web pages. The Page Digest was designed to support efficient document processing in applications such as Web change monitoring (Buttler et al., 2004; Buttler et al., 2003). XPack is an extension of this work that targets XML documents, in particular, by providing a more flexible framework that supports containerized document compression and efficient path querying.

The foundation work for modern data compression was done by Ziv and Lempel (Ziv and Lempel, 1977), who proposed the idea of the dictionary compressor. Dictionary compressors operate by substituting repeated occurrences of a given string with a shorter sequence; the original file can be reconstructed by reversing the substitution. This technique and its variants have become integral components of many standard computing tools such as the Gzip (loup Gailly and Adler, 2004) file compression tool. For a more general introduction to data compression, we refer the reader to (Sayood, 2000; Witten et al., 1999).

Recently, there have been several efforts to design XML-specific compression algorithms. The first, XMill (Liefke and Suciu, 2000), was designed to promote standardized document storage and transmission formats while alleviating the concern of data expansion that is often the penalty of converting data to XML. The XMill compressor achieves this goal by creating a container for each document tag and placing the data values for each tag into the same container. The containers are then compressed using a standard dictionary compression library. The intuition behind this approach is that grouping values by their tag names would arrange repetitious sections of the document closer to each other, which would yield greater compression efficiency from the dictionary compressor. The fundamental problem with the XMill approach is its opaque nature: data compressed with the XMill compressor is only available for use after being decompressed, a costly overhead step that must be added to the overhead of parsing the text document.

The Millau (Girardot and Sundaresan, 2000) binary format—an extension to the WAP Binary XML Format (World Wide Web Consortium, 1999)—has been introduced for the efficient encoding and streaming of XML structure, particularly in a wireless environment. A technique using multiplexed hierarchical PPM models was introduced in (Cheney, 2001). It has been shown to be slower than XMill, but in some cases achieves a higher degree of compression.

There have been several recent efforts to provide query support over compressed XML documents, typically by making a trade-off between the degree of compression and support for queries. The first of these techniques, XGRIND (Tolani and Haritsa, 2002), compresses XML documents by using Huffman encoding for non-enumerated types. If a document conforms to a known DTD, additional compression may be achieved by encoding the enumerated types listed in the DTD. XGRIND supports exact match and range queries over the compressed XML document. Similarly, XPRESS (Min et al., 2003) maintains the original structure of each XML document to support path queries, but instead uses a technique called reverse arithmetic encoding for compressing labeled paths of the document. In experiments, the XPRESS system is shown to be faster than XGRIND for compression and query resolution. In (Buneman et al., 2003), the authors present an XML compression technique that supports path queries over the compressed XML. Their technique relies on the identification of shared subtrees across a single document.

3 XPACK DOCUMENT ENCODING

XPack is a document encoding and compression system designed to operate over XML data. In this section, we construct a formal model of an XML document to facilitate the explanation of the techniques that XPack employs. Using this document model, we define a set of *document operators*, which are reversible functions that transform the XML data to reduce representation redundancy and provide efficient access to interesting document components.

3.1 Reference Document Model and Design Concepts

We model an XML document as an ordered tree of nodes where each node has a name and optionally contains a namespace qualifier, attributes, and text content. More formally, an XML document D is a set of tags $\{t_1, \ldots, t_{2n}\}$; each tag t_i is a string of characters denoting the tag's name and value. We say that tag t_i is a *closing tag* if the tag name begins with the slash character '/'; otherwise the tag is an *opening tag*. A document D is required to have the same number of opening and closing tags, which occur in tag pairs.[1] A tag pair describes a $node$ used in a document's tree model: a node is a descendent of the tag pairs that enclose it and an ancestor of the tag pairs that it encloses. The document D's tag set $\{t_1, \ldots, t_{2n}\}$ must satisfy the following invariants:

1. $\forall$ opening tags t_i, $\exists\ t_j$ s.t.
   (a) $i < j$
   (b) t_j is a closing tag for t_i
   (c) $\nexists$ a tag pair t_k, t_l s.t. $i < k < j < l$, and
   (d) $\nexists$ a tag pair t_k, t_l s.t. $k < i < l < j$
2. D contains the same number of opening and closing tags
3. t_1 is an opening tag
4. by extension of the above, t_{2n} is the closing tag for t_1

[1]Actual XML documents can also have "self closing" tags which combine the opening and closing tag. We model such tags as a tag pair t_i, t_{i+1}; in the ASCII version the closing tag t_{i+1} is implied.

Invariants 1 and 2 require all documents to contain a balanced tag set, i.e. each open tag must have a corresponding close tag. (a) and (b) state that open tags are required to preceed their closing tag. (c) and (d) are concerned with the proper nesting of tags; Figure 1 demonstrates proper nesting along with two examples of improper nesting. Finally, invariants 3 and 4 state that the first tag must be an open tag and that the last tag must be the corresponding closing tag.

```
...          ...          ...
<i>          <k>          <i>
  <k>          <i>          <k>
</i>         </k>           </k>
  </k>         </i>       </i>
...          ...          ...
(a)          (b)          (c)
```

Figure 1: Examples of improperly nested tags (a) and (b) along with a properly nested example (c).

3.2 XPack System Overview

The XPack encoding system operates over this document model to produce an XPack-encoded version of the document that retains the same structural elements and semantic meaning as the original document represented in a more efficient format. XPack espouses a container-oriented document structure that is created and modified by a set of unary document operators:

- PagePack (ϕ): document container encoding
- PathPack (ψ): path structure encoding
- NamePack (ρ): node tag name encoding
- URLPack (γ): document URL encoding
- AttributePack (α): attribute encoding
- ContentPack (χ): content encoding

XPack's document operators are designed to support flexible redundancy reduction. The PagePack operator is unique in that it operates over the original XML, while the remaining operators take a document that has already been containerized as their input. PagePack's purpose is to transform the text-oriented representation of an XML document into a compact tree-oriented representation of the document's structure augmented by a series of content containers. These containers can then be transformed, augmented, or replaced by subsequent operators. To as great an extent as possible, the remaining operators are designed to work in parallel so that overlapping computation can be used on parallel machines.

Figure 2 shows the XPack document compression process. When a document enters the system, a type detector module determines the document's type and loads a document type profile. The document type profile determines the default set of operators XPack will use in the redundancy elimination phase and also specifies how the document is split into components. Next, the document enters the redundancy elimination phase, which uses the selected XPack document operators to reduce the redundancy, and therefore the size, of the document. The minimized document components are then passed to the aggregator for reassembly and compressed to yield the final XPack-encoded document.

The heart of the XPack system is the redundancy elimination operators. The PathPack operator tries to reduce the space consumed by the document's tree structure. This encoding works best on documents that utilize structure more than content to convey meaning. PathPack is also designed to provide efficient access to the paths between document nodes for faster path matching and query operations. The NamePack operator utilizes observations about the tag names in XML documents to reduce their size. The URLPack operator reduces the space consumed by a document's URLs. The AttributePack operator combines the concepts found in the NamePack and URLPack operators and reduces the space consumed by element attribute names and values. AttributePack performs identifier substitution on attribute names and substitution and prefix production on attribute values.

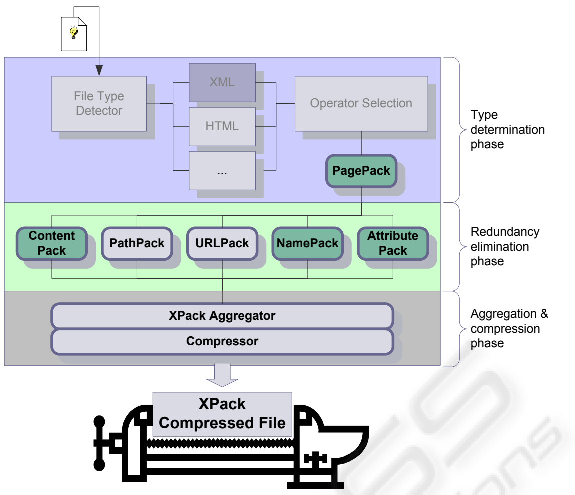

Figure 2: XPack system overview.

PagePack The PagePack operator ϕ creates a *node structure container* S by assigning a unique identifier to each node in the document and extracting the structure information inherent in the opening and closing tag sequence. $\phi(D) = (S, M)$ is a reversible function mapping an XML document D to a compact tree-structure representation and a set of containers for the document's content. $S = \{a_1, \ldots, a_n\}$ where a_i is the number of child nodes under the opening tag t_i. S preserves the document's tree structure by recording the relationship between opening tags. Node content is placed into a list of containers $M = \{m_1, \ldots, m_n\}$, which retain the information from the original document regarding each node's name, associated attributes, and content. The PagePack transformation stands as the basis upon which the other document operators are constructed.

Conceptually, PagePack encodes the document in three steps. First, the document tree is traversed in depth-first order and each node in the tree is assigned a unique identifier. For the sake of convenience, we choose an identifier equal to the visit order of each node; the root of the tree is assigned identifier "1." After each node has an ID, PagePack constructs a structure array that succinctly encodes the relationships between document nodes. Finally, node containers are constructed for each node in the document.

The remaining containers in the PagePack structure hold the information contained in the original nodes of the document tree, such as each node's tag name, attributes, and any associated content. Node containers are stored in index order, so the first node container holds the data belonging to the node with index "1," the root node.

PathPack Path structure encoding transforms the tree structure of an XML document into a sequence that encodes the paths found in the document tree to support efficient execution of path-style queries. Given a node structure container S, PathPack $\psi(S)$ is a reversible function that generates a sequence $x_j, 1 \leq j \leq m$, where $m \leq n$ and each x_j is a subpath tuple of the form $< start_j, end_j, parent_j >$. Each subpath in the sequence represents a nonbranching fragment of a root-leaf path; later paths in the sequence are further to the right and further down the tree than earlier paths in the sequence. For example, the subpath $< 2, 3, 1 >$ represents a two-node nonbranching path between nodes 2 and 3; the start node of path, 2, is a child of node 1. Given a node structure container S for document D, the following algorithm sketch highlights the main components of the PathPack operation:

1. set the start of the next path to the root node $a_1 \in S$
2. for each node $a_j \in S$
   (a) if a_j is a branch or leaf, output the current path. for each child c, set c to the start of the next path and run PathPack with it as the root.
   (b) otherwise continue.

NamePack The NamePack operator ρ eliminates document tag name redundancy by storing each unique tag name once and assigning a short reference to each name. For a document D with node container $M = \{m_1, \dots, m_n\}$, $\rho(M) = (I, M')$ is a reversible function that generates a set of tag name identifiers $I = \{i|$ unique tag names in $M\}$, stored in lexical order for efficient tag name searching.

NamePack reduces the redundancy of a Web document by generating tag name references and substituting the shorter references for the occurrences of the name in the document, eliminating the extra characters needed to store long and repeated tag names. NamePack is effective because the opening and closing tags that are the main structural feature of XML documents require tag names to be repeated; this necessity stems from the design of the document storage model but is not fundamental to representing structured documents in a computer system. Repetition of tag names can amount to a considerable proportion of the document's size.

NamePack collects the tag names from the node container M and stores each unique name in a new container. Occurrences of the names in the document are then replaced with a name reference that indicates which container and what name from that container is being referenced. M' is the new node container for D where tag names have been replaced with the appropriate index into the tag name container I.

```
<env:Envelope
  xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <n:alertcontrol
    xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2pm</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

Figure 3: Example SOAP Message

URLPack A common feature found in many Web documents is the hyperlink reference, which directs the application processing the document to further information or provides additional material for an end-user to explore. Hyperlinks are described via a URL that typically specifies a protocol, a target site, and a document reference. Figure 3 shows an example of a SOAP message used to invoke a remote Web service. SOAP messages make heavy usage of namespace references, which are externally linked documents that define the semantics of the call. The example message contains three such references.

We have observed that many URLs contain common prefixes. For example, two of the URLs in the SOAP message in Figure 3 start with the prefix "http://example.org/alert" and all three have the same "http://" protocol specifier. The URLPack operator γ is designed to eliminate repetition in a Web document's URLs by factoring out these common prefixes. URLPack uses a modified dictionary substitution mechanism (Bharat et al., 1998) for encoding URLs that is restricted to start of string prefixes to maximize efficiency.

In general, the URLPack operator $\gamma(M) = (U, M')$ where $U = \{u|u \in extractURLs(M)\}$; $extractURLs$ encodes the document's URLs through the following steps:

1. Retrieve the document's URLs from the node container M
2. Sort the URLs into lexical order, remove duplicates
3. For each u_i, replace the prefix shared with the expanded form of u_{i-1} with the count of shared characters

AttributePack The AttributePack operator α is a logical combination of the ideas found in the NamePack and URLPack operators that is used to compress document attributes and expose them for faster processing. Attributes are associated with nodes: a single node can have zero or more attributes, each of which consists of a name and a possibly empty value. Attribute names are frequently reused throughout the document; certain attribute values—hyperlink reference URLs, for instance—will also appear multiple times in a document's attributes.

Consistent with the approach we have espoused with the other XPack operators, AttributePack extracts attribute names and values from a document D's node containers M. For a document D with node container $M = \{m_1, \dots, m_n\}$, the AttributePack operator $\alpha(M) = (X, Y, M')$ where $X = \{x|$ unique attribute names in $M\}$, stored in lexical order for efficient searching, and $Y = \{y|y \in extractAttributeValues(M)\}$; $extractAttributeValues$ operates identically to the function $extractURLs$ used in the URLPack operator γ.

ContentPack The ContentPack operator χ eliminates duplication of document content and organizes the content to achieve better document compression. ContentPack $\chi(M) = (C, M')$ is a reversible function mapping a document D's node containers M to a content list C and an updated list of node containers M'. The updated node containers replace each node's content with a reference to the appropriate entry in the content list for that node. Any duplicated text nodes in the original document will receive references to the same entry in C. It is common for ASCII-encoded XML documents to contain many equivalent whitespace nodes for formatting purposes to assist human readability of the document. ContentPack eliminates this and any other content redundancy that is present in the document's non-markup content.

4 EXPERIMENTAL EVALUATION

The experiments in this section are designed to test the features of the XPack system. Due to space limitations, we restrict our experimental evaluation to two sets of experiments that are designed to provide a representative sample of XPack's performance characteristics. The first set of experiments demonstrate the compression performance of XPack over several document sets. The second set of experiments test the query performance of XPack and demonstrate that it is possible to achieve both good compression and performance. The experiments show the power of the XPack encoding system: many interesting document operations can be completed with only a partial decompression of the document.

4.1 Experimental Environment

To test the XPack design experimentally, we have implemented a prototype XPack encoder and query processor. The prototype is implemented in Java. All experimental results presented here were conducted on a PC with an AMD Athlon XP processor at 1.4GHz with 512MB RAM running Windows XP Professional. The experiments were run on Sun's Java virtual machine for Windows, version 1.4.2. The Apache Foundation's Xerces XML parser was used for testing SAX and DOM document materialization performance and the Xalan XSLT library provided XPath query evaluation support for DOM documents.

4.1.1 Experimental Data

Random Walk. The random walk data set consists of approximately 2000 Web pages gathered by an automated crawler. To gather the data, the crawler's URL frontier was seeded with a small set of start pages. At each iteration, a URL was chosen at random from the frontier and the corresponding document retrieved. The document was then converted from HTML into XML and annotated with a single comment tag recording the originating URL of the document and a timestamp. The document's links were then added to the URL frontier for possible selection in the next iteration. The average size of the documents collected after normalization to XML was 43,888 bytes with a minimum of 161 bytes, maximum of 898,924 bytes, and standard deviation of 37,150 bytes.

Shakespeare. The Shakespeare data set is a public domain collection of Shakespeare's plays that have been converted into XML. The tag set is small and consists of such tags as LINE, SPEECH, and SPEAKER. None of the nodes have attributes. The data set contains 37 documents whose average size is 213,448 bytes with a minimum of 141,345 bytes, maximum of 288,735 bytes, and standard deviation of 36,345 bytes.

4.2 XPack Compression Performance

Our first set of experiments test the aggregate performance of the XPack compression system. The data sets that we have selected are intended to represent a broad cross-section of the types of XML documents typically used by applications. The random walk data set contains documents representative of the XHTML format used by modern, standards compliant Web applications. An important characteristic of this data set is the large number of attributes containing long, somewhat similar values. In contrast, the Shakespeare data set is characterized by a large amount of text content with no attributes and a small, terse tag set.

The documents in these experiments were converted from XML to three compressed document formats: gzip compressed XML, XMill, and XPack. The XPack documents were encoded with the PagePack, NamePack, and AttributePack redundancy elimination operators. The resulting XPack document containers, including the node containers holding the text content of the document, were then sent through a gzip compression filter and written to disk.

The random walk data set contains documents having a relatively small tag dictionary consisting of short names—each document employs a subset of the HTML tag set. These documents contain many attributes with repeated or similar values, such as hyperlink URLs. The documents also have significant text content but are not dominated by it as in the Shakespeare data set. Figure 4 demonstrates the effectiveness of the XPack encoding system for compressing the random walk data set. The graph shows the size of the XMill, gzip, and XPack compressed versions of the documents, whose original sizes appear on the x-

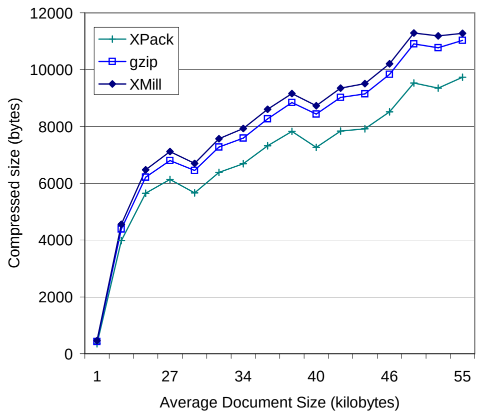

Figure 4: XPack document compression, random walk data set.

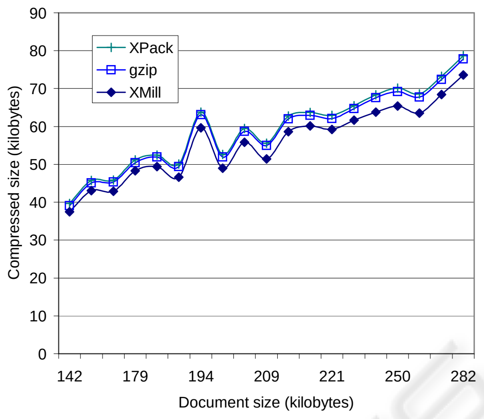

Figure 5: XPack document compression, Shakespeare data set.

axis. The documents have many repetitive attributes, such as navigation URLs, that provide a significant opportunity for redundancy elimination by AttributePack, allowing XPack to achieve compression rates up to 20% better than the other systems' rates.

In contrast with the random walk dataset, the Shakespeare data set is content-heavy, with the majority of the size of the documents occupied by actors' lines. None of the documents contained node attributes, eliminating any potential savings from AttributePack. The tag dictionaries are composed of a few short names. Even in the Shakespeare data set, with its limited redundancy, XPack achieves compression rates comparable to the other systems. Compression results for the Shakespeare data set are presented in Figure 5, which demonstrates that XPack can achieve compression rates comparable to XMill and gzip. XPack enjoys a significant usability advantage over the systems due to its query-capable format.

4.3 Query Performance Experiments

The next experiment evaluates the XPack system's ability to provide query support over compressed documents by testing the performance of XPath path expression queries on XPack documents. Path queries over XML documents using the XPath node selection language are a popular means for extracting data from XML documents. Table 1 presents the performance results obtained for two test queries over the Shakespeare data set and compares these results with the performance of the same queries over standard and compressed XML. The first query, "//PERSONA," selects all the nodes in the document with the name "PERSONA" while the second selects all SCENE nodes with a parent ACT and a root grandparent PLAY. For each of the three document types used in the experiment—DOM, compressed DOM, and XPack—Table 1 lists the frequencies of query execution times obtained executing the query over each of the 37 documents in the Shakespeare data set. Note that these are "cold start" measurements: the time recorded for each test is the sum of the time needed to load the document from disk, parse it into an internal memory representation, and execute the query.

Table 1: XPath Evaluation Time, Shakespeare data set.

//PERSONA

Time (ms)	DOM	gzDOM	XPack
≤ 100	0	0	15
101–200	0	0	22
201–500	0	0	0
501–600	28	8	0
601–700	9	29	0

/PLAY/ACT/SCENE

Time (ms)	DOM	gzDOM	XPack
≤ 100	0	0	15
101–200	0	0	22
201–500	0	0	0
501–600	29	5	0
601–700	8	32	0

These results indicate that XPack can support efficient evaluation of path queries while simultaneously reducing the storage and materialization costs for XML documents. For many types of interesting queries, XPack's compartmentalization and separation of document components supports faster querying than the original documents: as shown in Table 1, document queries requiring more than 500 ms using a standard XML parsing and querying library can be executed in less than 200 ms with XPack. Because the queries in question require only the document structure and tag names to be satisfied, XPack can service the query without reading the entire document, which is an impossibility with XML. These measurements also demonstrate XPack's advantage over opaque XML compression systems like gzip and XMill, which must add document decompression overhead to the cost of a complete parse of the document.

5 CONCLUSION

The XPack document encoding and compression system achieves both good compression and strong query support for compressed documents. *Redundancy elimination* allows XPack to significantly reduce the size of Web documents, while *compartmentalization* separates the components of a document into logical containers. XPack's binary encoding scheme eliminates the expense of parsing XML text into memory objects by instead storing the document in a format that can be read directly into memory and immediately operated upon. XPack's compression performance compares favorably with other widely used XML compression systems in testing with document sets having considerably different structural and content characteristics. XPack's document compartmentalization enables efficient document querying using XPack's aggregate document operators and the more general XPath query mechanism, which enjoys up to a 95% performance increase over queries using a standard XPath processor and text-based XML.

REFERENCES

Altinel, M. and Franklin, M. J. (2000). Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB '00)*.

Bharat, K., Broder, A., Henzinger, M., Kumar, P., and Venkatasubramanian, S. (1998). The connectivity server: Fast access to linkage information on the web. In *Proceedings of the Seventh International World Wide Web Conference (WWW '98)*.

Bray (1998). Extensible markup language (XML) 1.0. Technical report, W3C.

Buneman, P., Grohe, M., and Koch, C. (2003). Path queries on compressed xml. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB '03)*.

Buttler, D., Liu, L., and Rocco, D. (2003). Efficient processing of web page sentinels using page digest. Technical report, Georgia Institute of Technology.

Buttler, D., Rocco, D., and Liu, L. (2004). Efficient web change monitoring with page digest. *13th Annual International World Wide Web Conference WWW2004 (poster symposium)*.

Cheney, J. (2001). Compressing XML with multiplexed hierarchical PPM models. In *Data Compression Conference*.

Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web services description language (WSDL) 1.1. Technical report, W3C.

Girardot, M. and Sundaresan, N. (2000). Millau: an encoding format for efficient representation and exchange of xml over the web. In *Proceedings of the Ninth International World Wide Web Conference (WWW 2000)*.

Liefke, H. and Suciu, D. (2000). XMill: an efficient compressor for XML data. In *ACM International Conference on Management of Data (SIGMOD)*, pages 153–164.

loup Gailly, J. and Adler, M. (2004). Gzip compression algorithm. http://www.gzip.org/algorithm.txt.

Min, J.-K., Park, M.-J., and Chung, C.-W. (2003). Xpress: A queriable compression for xml data. In *Proceedings of the 2003 ACM Conference on Management of Data (SIGMOD '03)*.

Mitra, N. (2003). Soap version 1.2 part 0: Primer. Technical report, World Wide Web Consortium.

Rocco, D., Buttler, D., and Liu, L. (2003). Page digest for large-scale web services. In *Proceedings of the IEEE Conference on Electronic Commerce*.

Savourel, Y. (2001). *XML Internationalization and Localization*. SAMS.

Sayood, K. (2000). *Introduction to Data Compression*. Morgan Kaufmann, New York, second edition.

Tolani, P. and Haritsa, J. R. (2002). XGRIND: A query-friendly XML compressor. In *ICDE*.

Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, New York, second edition.

World Wide Web Consortium (1999). WAP binary XML content format.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(65):337–343.