A Compressor for Effective Archiving, Retrieval, and Update of XML Documents

JUN-KI MIN, MYUNG-JAE PARK, and CHIN-WAN CHUNG
Korea Advanced Institute of Science and Technology

Like HTML, many XML documents are resident on native file systems. Since XML data is irregular and verbose, the disk space and the network bandwidth are wasted. To overcome the verbosity problem, the research on compressors for XML data has been conducted. However, some XML compressors do not support querying compressed data, while other XML compressors which support querying compressed data blindly encode tags and data values using predefined encoding methods. As a result, the query performance on compressed XML data is degraded. Furthermore, existing XML compressors do not provide the facility for updates on compressed XML data.

In this paper, we propose XPRESS, an XML compressor which supports direct updates and efficient evaluations of queries on compressed XML data. XPRESS adopts a novel encoding method, called reverse arithmetic encoding, which is intended for encoding label paths of XML data, and applies diverse encoding methods depending on the types of data values. Experimental results with real-life data sets show that XPRESS achieves significant improvements on query performance for compressed XML data and reasonable compression ratios. On the average, the query performance of XPRESS is 2.99 times better than that of an existing XML compressor and the compression ratio of XPRESS is about 75%. Additionally, we demonstrate the efficiency of the updates performed directly on compressed XML data.


General Terms: Algorithms, Management, Performance

Additional Key Words and Phrases: Compression, Query Processing, XML

1. INTRODUCTION

The eXtensible Markup Language (XML) [Bray et al. 1998] is intended as a markup language for an arbitrary document structure, as opposed to HTML which is a markup language for a specific kind of hypertext data.

XML data comprises hierarchically nested collections of elements, where each element is represented by a start tag and an end tag that describe the semantics...
of the element. Using tags, XML separates contents and the representation (i.e., structure) in XML documents. In addition, an element in XML data can contain either atomic raw character data or a sequence of nested subelements and can have a number of attributes composed of name-value pairs.

The basic data model of XML is a labeled tree, where each element or attribute is represented as a node in the tree, and its tag corresponds to the label of the corresponding node. This tree structured data model is simple enough to devise efficient as well as elegant algorithms for it. Due to its flexibility and simplicity, XML is rapidly emerging as the de facto standard for exchanging and querying documents on the web required for the next generation web applications including electronic commerce and intelligent web searching.

To retrieve XML data, XML query languages such as XPath [Clark and DeRose 1999] and XQuery [Boag et al. 2002] have been proposed recently. These languages are based on path expressions to traverse irregularly structured data. Therefore, the efficient support of path expressions over XML data is a major issue in the field of XML [Goldman and Widom 1997; Grust 2002].

Currently, a variety of research for XML data has focused on issues related to XML storage [Florescu and Kossman 1999], retrieval [Goldman and Widom 1997; Fernandez and Suciu 1998], and publication [Fernandez et al. 2000; Shanmugasundaram et al. 2000]. Although some XML data are managed in the XML storage, large portions of XML data are still on native file systems as in the case of HTML. Thus, in order for XML to become the true internet standard, the research on the efficient management of the file based XML data is required.

One of the interesting applications for file based XML data is web searching. In this application, if each web server manages its own data in the form of XML and transmits it through the network, the storage and the network bandwidth are wasted since XML data is irregular and verbose. To overcome the verbosity problem, the research on compressors for XML data has been conducted [Liefke and Suciu 2000; Tolani and Haritsa 2002].

XMill [Liefke and Suciu 2000] was designed to minimize the size of compressed XML data. However, XMill was not intended to support querying compressed XML data.

Recently, XGrind [Tolani and Haritsa 2002] was devised to evaluate queries directly on compressed XML data. However, the encoding scheme of XGrind does not sufficiently take account of the properties of XML data and query languages.

Furthermore, XMill and XGrind do not support direct updates on compressed XML data. Thus, to update the XML data, the compressed XML data should be decompressed completely. And then, updates and recompression are performed. This approach consumes much time.

1.1 Our Contributions

In this paper, we propose XPRESS, an XML compressor, to compress XML data for the purposes of archiving, retrieving and exchanging. XPRESS supports direct updates and efficient evaluations of queries on compressed XML data.

In contrast to the web search engines for HTML, XML search engines can use structural predicates such as path expressions for search conditions since XML differentiates the structure from contents. For example, if users want to select
XML files that contain some information about sales of houses, users can submit a search condition like "∃(// sales/house)".

To perform the kinds of queries as mentioned above on the compressed data in XMill, a complete decompression is required. In XGrind, although the overhead for the complete decompression is removed, the overhead of maintenance and evaluation of the simple path to each element, similar to that for uncompressed XML data, still remains. In contrast to the other XML compressors, XPRESS gets rid of this overhead by using a novel encoding method, called reverse arithmetic encoding, and minimizes the overhead of partial decompression by utilizing diverse encoding methods.

Existing XML compressors do not provide the facility for update on compressed XML data. To our best knowledge, XPRESSS is the first XML compressor which supports direct update on compressed XML data.

XPRESS has the following novel combination of characteristics to compress, retrieve, and update XML data efficiently.

— **Reverse Arithmetic Encoding:** Since existing XML compressors simply represent each tag by using a unique identifier, they are inefficient to handle path expressions on compressed XML data. In contrast, XPRESS adopts the reverse arithmetic encoding method that encodes a label path as a distinct interval in [0.0, 1.0). Using the containment relationships among the intervals, path expressions are evaluated on compressed XML data efficiently.

— **Automatic Type Inference:** Some XML compressors compact data values of XML elements by using predefined encoding methods (e.g., huffman encoding). However, according to the types of data values, the kinds of efficient encoding methods are different. In some XML compressors, the types of data values are manually interpreted. Thus, if there is no human interference, data values of XML elements and attributes are not compressed properly. In XPRESS, to apply effective encoding methods to various kinds of data values of XML elements, we devise an efficient type inference engine that does not require the human interference.

— **Apply Diverse Encoding Methods to Different Types:** According to the inferred type information, we apply proper encoding methods to data values. Thus, we achieve a high compression ratio and minimize the overhead of partial decompression in the query processing phase.

— **Support of Direct Update:** To update compressed XML data, existing XML compressors should perform the complete decompression. But, by analyzing the portion of the XML data to be inserted, XPRESS performs the partial decompression of the compressed XML data. Thus, XPRESS supports direct updates on the compressed XML data without the complete decompression and recompression.

— **Semi-adaptive Approach:** Our compression scheme is categorized as the semi-adaptive approach [Howard and Vitter 1991] which uses a preliminary scan of the input file to gather statistics. Since the semi-adaptive approach does not change the statistics during the compression phase, the encoding rules for data are independent to the locations of data. This property allows us to query compressed XML data directly.
Homomorphic Compression: Like XGrind, XPRESS is a homomorphic compressor which preserves the structure of the original XML data in compressed XML data. Thus, XML segmentations that satisfy given query conditions are efficiently extracted.

We implemented XPRESS and conducted an extensive experimental study with real-life XML data sets. In our experiment, XPRESS demonstrates significantly improved query performance and reasonable compression ratio compared to the other XML compressors. In addition, to show the efficiency of the updates, we compared the update performance of XPRESS with a naive approach. On the average, the query performance of XPRESS is 2.99 times better than that of an existing XML compressor and the compression ratio of XPRESS is 75%.

1.2 Organization
The remainder of the paper is organized as follows. In Section 2, we present general purpose compression methods and compression tools for XML data. In Section 3, we present the features of XPRESS. Section 4 describes the compression techniques of XPRESS. Section 5 provides the update processing technique of XPRESS in detail. Section 6 contains the result of our experiments which compares the performance of XPRESS to those of the other XML compressors. Finally, in Section 7, we summarize our work and suggest some future studies.

2. RELATED WORK
The data compression has a long and rich history in the field of information theory [Shannon 1948; Huffman 1952].

One advantage of data compression is that the required disk space of data can be reduced significantly. The second advantage is the saving of the network bandwidth. Since the overall size of data is decreased, much more data can be transferred through the network within a given period of time. Another advantage is that data compression improves the overall performance of database systems. By compressing data, more information can be loaded in the buffer and the number of disk I/Os is reduced. Therefore, the performance of the system is enhanced.

2.1 General Purpose Compression
According to the ability of data recovery, compression methods are classified into two groups: the lossy compression and the lossless compression.

The lossy compression reduces a file by permanently eliminating certain information. The data compressed by the lossy compression cannot be reconstructed into the original data by the decompression. Thus, in this paper, we do not address the lossy compression since the lossless recovery is required for textual information.

The lossless compression is categorized into three groups: static, semi-adaptive, adaptive [Howard and Vitter 1991]. The static compression uses fixed statistics or does not use any statistics. The semi-adaptive compression scans the input data to gather statistics preliminarily and rescans the data to compress. The adaptive compression does not require any prior statistics. Instead, statistics are gathered dynamically, and updated during the compression phase.
The representative compression methods of the static compression are dictionary encoding, binary encoding and differential encoding.

The dictionary encoding method assigns an integer value to each new word from the input data so that each word in the input data can be compressed by using a uniquely assigned integer value. Some special types of data such as numeric data can be encoded in binary, e.g., integer or floating. This is called the binary encoding method. The differential encoding method, also called delta encoding, replaces a data item with a code value that defines its relationship to a specific data item. For example, a data sequence of 1500, 1520, 1600, 1550 will be encoded as 1500, 20, 100, 50.

Since the static approach does not consider the nature of given data, compression ratios are quite different depending on the input data. Thus, it is important to adopt proper encoding methods on account of data properties.

In the semi-adaptive compression methods, huffman encoding [Huffman 1952] and arithmetic encoding [Witten et al. 1987] are the examples.

The basic idea of the huffman encoding method is to assign shorter codes to more frequently appearing symbols and longer codes to less frequently appearing symbols. To assign a code to each character, a binary tree, called the huffman tree, is constructed using the statistics gathered by a preliminary scan. A simple example of the huffman tree is shown in Figure 1. The leaf nodes of the huffman tree are assigned symbols in input data. The value in a leaf node is the frequency of the symbol. The left edges of the huffman tree are labeled with 0 and the right edges are labeled with 1 so that the code assigned to each symbol is the sequence of labels starting from the root to the leaf node of the symbol.

The arithmetic encoding method represents a given message by choosing any number from a calculated interval. Symbols are assigned disjoint intervals according to their frequencies. Successive symbols of a message reduce the length of the interval of the first symbol in accordance with the frequencies of the symbols. After reducing the length of the interval by applying all the symbols of the message, the message is transformed into a variable length bit string that represents any number within the reduced interval.

In the adaptive compression, adaptive huffman encoding, adaptive arithmetic encoding, and LZ encoding are representatives.

The adaptive huffman encoding method and the adaptive arithmetic encoding method are similar to the huffman encoding method and the arithmetic encoding method.
method, respectively. However, the adaptive huffman encoding method dynamically construct the huffman tree, and the adaptive arithmetic encoding method calculates the intervals by gathering frequencies and probabilities dynamically. In other words, these adaptive encoding methods dynamically update statistics (i.e., huffman tree or intervals) of each symbol based on the previous statistics during compression phase, instead of using the predefined statistics used in the semi-adaptive compression methods.

The LZ (Lempel-Ziv) encoding method, similar to the dictionary encoding method of the static compression, records the string seen previously. When the new string is read, the LZ encoder finds the longest common substring in the string seen previously, then converts the new string into a pointer to the common substring with an additional string. (see more details in [Salomon 1998]).

2.2 XML Compression

Recently, some research on compressors for XML data have been conducted. The representative XML compressors are XMill and XGrind.

XMill physically separates XML tags and attributes from their data values and groups semantically related data values into containers. XML tags and attributes are compressed by the dictionary encoding method. Each container can be compressed by a user specified encoding method. In order to apply specialized compressors to containers, it needs the interpretations of containers from human. Finally, each compressed container is recompressed by a build-in library, called zlib. Since XMill does not maintain the original structure of XML data on compressed XML data by physically separating the structure from data values and grouping semantically related data values, XMill does not support direct querying on compressed XML data.

A distinguishable feature of XGrind compared to XMill is that it supports querying compressed XML data. In XGrind, data values are compressed by huffman encoding or dictionary encoding and tags are compressed by dictionary encoding. Using DTD, XGrind determines to apply huffman encoding or dictionary encoding for a certain attribute value. In XGrind, to evaluate a path expression, whenever an element is visited by the query processor, the identifier sequence which represents the label path from the root element to the currently visited element is found and the query processor checks whether this identifier sequence satisfies the path expression. In addition, to evaluate range queries on compressed XML data, a partial decompression is always required since huffman encoding and dictionary encoding do not preserve any order information among data items.

Furthermore, none of them do support direct updates on compressed XML data. Thus, to update XML data, a complete decompression and recompression should be required.

3. FEATURES OF XPRESS

In this section, we present the major features of XPRESS which support effective query processing on compressed XML data. In our work, we do not distinguish attributes from elements since attributes in XML data are considered as specific elements.
To support an effective evaluation of path expressions, we devise a novel encoding method, called reverse arithmetic encoding, which is inspired by arithmetic encoding. We first define some notations with a simple XML data to explain our proposed encoding method.

\textbf{Definition 1.} A simple path of an element $e_n$ in XML data is a sequence of one or more dot-separated tags $t_1, t_2 \ldots t_n$, such that there is a path of $n$ elements starting from the root element $e_1$ to $e_n$ and the tag of the element $e_i$ is $t_i$.

For example, in the XML data shown in Figure 2, the simple path of a section element is \texttt{book.section}.subsection.

\textbf{Definition 2.} When the simple path of an element $e$ in XML data is $a_1, a_2 \ldots a_n$, a dot-separated tag sequence $b_k, b_{k+1} \ldots b_n$ is a label path of $e$ if we have $b_k = a_k$, $b_{k+1} = a_{k+1} \ldots b_n = a_n$, where $1 \leq k$ and $k \leq n$. Furthermore, for two label paths, $P = p_i \ldots p_n$ and $Q = p_j \ldots p_n$ of $e$, if $i \geq j$, then we call $P$ is a suffix of $Q$.

Again in Figure 2, section.subsection is a label path of the subsection element. And, subsection is a suffix of section.subsection. In XML, the structural constraints of queries are based on the label path such as //section/subsection.

Now, we present the reverse arithmetic encoding method. In contrast to existing XML compressors that transform the tag of each element to an identifier, reverse arithmetic encoding represents the simple path of an element by an interval of real numbers between 0.0 and 1.0. The basic idea of reverse arithmetic encoding is simple but elegant.

First, reverse arithmetic encoding partitions the entire interval $[0.0, 1.0)$ into subintervals, one for each distinct element (in contrast to one of multiple elements with same tag). An interval for element $T$ is represented as $\text{Interval}_T$. The size of $\text{Interval}_T$ is proportional to the frequency (normalized by the total frequency) of element $T$. The following example shows the intervals for elements in Figure 2.

\textbf{Example 1.} Suppose that the frequencies of elements = \{book, author, title, section, subsection, subtitle\} are \{0.1, 0.1, 0.1, 0.3, 0.3, 0.1\}, respectively. Then, based on the cumulative frequency, the entire interval $[0.0, 1.0)$ is partitioned as follows:
Next, reverse arithmetic encoding encodes the simple path \(P = p_1 \ldots p_n\) of an element \(e\) into an interval \([\text{min}_e, \text{max}_e]\) using the algorithm in Figure 3.

Intuitively, the function reverse_arithmetic_encoding reduces Interval \(p_n\) using the interval for the simple path \(p_1 \ldots p_{n-1}\) where \(p_n\) is the tag of the element \(e\). For understanding, we used a recursive function call in Line (4) of Figure 3. Basically, we encode the simple path of an element in a given XML data to an interval starting from the root element to other elements in the depth first tree traversal. Therefore, the recursion is not necessary in implementation since \([q_{\text{min}}, q_{\text{max}}]\) has already been computed at the time of encoding the parent element of \(e\). Thus, the time complexity to compute all intervals of elements can be easily shown to be \(O(E)\), where \(E\) is the number of elements in a given XML data.

### Function reverse_arithmetic_encoding

\[
\text{begin} \\
1. \quad [\text{min}_e, \text{max}_e] := \text{Interval}_p \\
2. \quad \text{if}(n = 1) \quad \text{return} \quad [\text{min}_e, \text{max}_e] \\
3. \quad \text{length} := \text{max}_e - \text{min}_e \\
4. \quad [q_{\text{min}}, q_{\text{max}}] := \text{reverse_arithmetic_encoding}(p_1 \ldots p_{n-1}) \\
5. \quad \text{min}_e := \text{min}_e + \text{length} \times q_{\text{min}} \\
6. \quad \text{max}_e := \text{min}_e + \text{length} \times q_{\text{max}} \\
7. \quad \text{return} \quad [\text{min}_e, \text{max}_e] \\
\text{end}
\]

Fig. 3. An algorithm of reverse arithmetic encoding

Example 2 which is the continuation of Example 1 illustrates the behavior of reverse arithmetic encoding.

**Example 2.** The interval \([0.69, 0.699)\) for a simple path book.section.subsection in Figure 2 is obtained by the following process:

<table>
<thead>
<tr>
<th>element</th>
<th>simple path</th>
<th>Interval</th>
<th>subinterval</th>
</tr>
</thead>
<tbody>
<tr>
<td>book</td>
<td>book</td>
<td>(0.0, 0.1)</td>
<td>(0.0, 0.1)</td>
</tr>
<tr>
<td>section</td>
<td>book.section</td>
<td>(0.3, 0.6)</td>
<td>(0.3, 0.33)</td>
</tr>
<tr>
<td>subsection</td>
<td>book.section subsection</td>
<td>(0.6, 0.9)</td>
<td>(0.69, 0.699)</td>
</tr>
</tbody>
</table>

In the aspect of the utilization of the intervals, the reverse arithmetic encoding method is similar to the region numbering scheme originated in the field of information retrieval (IR) [Salminen and Tompa 1992]. Some XML storage systems [Li and Moon 2001] utilize the region numbering scheme to denote XML elements and attributes. The intervals (i.e., regions) generated by the region numbering scheme represent the relationship among elements (e.g., ancestor and descendant relationships). Suppose that there are two elements \(x\) and \(y\), where \(x\) is an ancestor of
y, then the corresponding intervals \((x_s, x_e)\) and \((y_s, y_e)\) for \(x\) and \(y\), respectively, satisfy the following property, that is \(x_s \leq y_s \leq y_e \leq x_e\).

In contrast to the region numbering scheme, the intervals generated by reverse arithmetic encoding express the relationship among label paths as follows:

**Property 1.** Suppose that a simple path \(P\) is represented as the interval \(I\), then all intervals for suffixes of \(P\) contain \(I\).

For instance, the interval \([0.6, 0.9)\) for a label path `subsection` and the interval \([0.69, 0.78)\) for a label path `section.subsection` contain the interval \([0.69, 0.699)\) for a simple path `book.section.subsection`. If a label path expression of a query is `//section/subsection`, this label path expression is represented as an interval \([0.69, 0.78)\).

And then, the query processor efficiently selects the elements whose corresponding intervals are within \([0.69, 0.78)\). As a result, path expressions based on label paths are effectively evaluated by Property 1.

Finally, without any loss of information, the start tag of an element \(e\) is replaced by the minimum value of the subinterval generated by the function reverse_arithmetic_encoding. Since the minimum value of the subinterval is also consistent to Property 1, the corresponding tag of a minimum value can be obtained at the decompression phase easily using binary lookup of `IntervalTs`. In addition, path expressions are evaluated at the query processing phase effectively.

Furthermore, reverse arithmetic encoding can be naturally applied to some XML storage systems [Shimura et al. 1999; Tatarinov et al. 2002] which maintain the path information of individual elements by the path identifier.

Our encoding scheme belongs to the semi-adaptive compression. Since statistics, required in the XML compression phase, are collected and fixed at the preliminary scan, the generated code by XPRESS is independent to the location of the corresponding symbol (tags and data values).

If the adaptive compression such as adaptive huffman encoding is applied, the compression time is saved since the preliminary scan is not required. However, in the adaptive compression, the encoded value of a certain symbol is changed depending on the location of the occurrence of the symbol since the adaptive compression modifies the encoding model (e.g., huffman tree) dynamically. Thus, to evaluate a query with data value predicates, the complete decompression of compressed XML data is required. This degrades the query performance severely. Note that, generally, the XML data compression is an one time operation and queries are evaluated repeatedly. Therefore, the two-scan overhead on the XML data compression is compensated by frequent query evaluations.

Also, at the preliminary scan, XPRESS infers the type of data values of each distinct element. As described in Section 2, depending on the type of data values, the effective data encoding methods are different. However, existing XML compressors blindly use predefined encoding methods or apply some encoding methods manually. For example, in XMill, data values are bypassed to a built-in compression library, `zlib`, if the data encoders are not specified manually. Additionally, in XGrind, the data values for elements and general attributes are compressed by huffman encoding and the data values of enumeration typed attributes are compressed by dictionary encoding. Without considering the nature of data values, the size of compressed XML data may increase. Therefore, we devise an effective type
inference engine which infers the type of data values of each distinct element by simple inductive rules during the preliminary scan phase.

In the compression phase, data values are compressed by proper encoding methods according to their inferred types. Although the huffman encoder and the dictionary encoder are effective to general textual data, these methods do not preserve the order relationship among data. That is, let two data be $v_1$, $v_2$ and their corresponding compressed version be $c_1$, $c_2$, then $v_1 > v_2 \not\Rightarrow c_1 > c_2$. Thus, to evaluate the queries with the range of data values, a partial decompression should be performed.

While, for numeric typed data values, XPRESS applies binary encoding first and then differential encoding with the minimum value. For example, data values of element e “120”, “150”, “100” “130” are transformed into integers 120, 150, 100, 130 and encoded as 20, 50, 0, 30. Since this encoding method preserves the order relationship among data values, the overhead of a partial decompression for numeric typed data is removed. However, XPRESS adopts the huffman encoder and the dictionary encoder for textual data since we can not find an effective encoding method which preserves the order relationship among textual data and achieves similar compression ratios compared to those of the huffman encoder and the dictionary encoder.

Like XGrind, XPRESS obeys the homomorphism [Tolani and Haritsa 2002]. The homomorphic compression technique preserves the structure of the original XML data on compressed XML data.

As shown in Figure 4-(b), some XML compression tools such as XMill physically separate structures (i.e., tag) and data (i.e., value). Here, the tags A and B are encoded as T1 and T2, respectively, and the end tags are replaced by '/'. By applying this technique, a built-in compression library such as zlib can reduce the size of compressed XML data well since the strings which have semantically/syntactically similar properties are grouped into a container. However, this technique incurs difficulty in query processing, also in performing the updates since the structure of compressed XML data is differentiated compared to the original XML data. For example, to compute a query “/A/B[text()='v2']” on compressed XML data in Figure 4-(b), two file pointers are needed to keep the currently visited locations of the first container (C0) and the second container (C1)\(^1\). If the structure of XML

---

\(^1\)each container is represented by a dotted box in Figure 4-(b)

ACM Transactions on Internet Technology, Vol. V, No. N, Month 20YY.
data and/or the predicates of queries are very complex, a dedicated handling of multiple file pointers is required. However, handling the multiple file pointers effectively is very hard and results in inefficient query processing. In contrast to the non-homomorphic compression, since the homomorphic compression preserves the structure of the original XML data, the homomorphic compression allows us to evaluate queries and extract XML segmentations which satisfy given query conditions efficiently. Also, the homomorphic compression allows us to perform the updates efficiently since the updating XML segmentations are easily applied on compressed XML data.

As a result, based on the above features, the compressed XML data generated by XPRESS supports the query processing and the updates effectively without the complete decompression of compressed XML data.

4. COMPRESSION TECHNIQUES IN XPRESS

In this section, we present the architecture of XPRESS and detailed techniques developed for XPRESS.

Based on the features described in Section 3, we designed the architecture of XPRESS as depicted in Figure 5.

![Fig. 5. The architecture of XPRESS](image)

The core modules of XPRESS are XML Analyzer and XML Encoder. As mentioned earlier, the compression scheme of XPRESS is categorized as the semi-adaptive compression. During the preliminary scan of given XML data, XML Analyzer (see details in Section 4.1) is invoked. XML Analyzer gathers the information used by XML Encoder (see details in Section 4.2) which generates queriable compressed XML data.

XML Analyzer consists of two submodules: the statistics collector and the type inference engine. The statistics collector computes the adjusted frequency (see Section 4.1) of each distinct element. The adjusted frequencies of elements are used as inputs to the reverse arithmetic encoder. The type inference engine infers the type of data values of each distinct element inductively and produces the statistics for the type dependent encoders in XML Encoder.

4.1 XML Analyzer

The main algorithm of XML Analyzer is shown in Figure 6.
To compute the frequency of each distinct element, the procedure `Statistics_Collection` is executed. To infer the types of data values, the procedure `Type_Inferencing` is executed. The algorithm `XML_Analyzer` generates a hash table called `Elemhash`. The entry of `Elemhash` is `ELEMINFO` which keeps the information (e.g., type of data values, frequency) of each distinct element. A stack called `Pathstack` is used to keep the trace of the currently visited element.

To get `Interval` for each distinct element, the statistics collector can simply count the number of occurrences of each distinct element. However, since tags of higher level elements (e.g., the root element) appear rarely, the intervals for simple paths shrink quickly. This requires the use of high precision floating arithmetic.

To prevent the rapid shrinking of an interval, we can use the concept of the path tree which is devised for the selectivity estimation of XML path expressions [Aboulnaga et al. 2001]. Every node in the path tree represents a simple path of XML data. The path tree of XML data in Figure 2 is shown in Figure 7-(a). In the original path tree of [Aboulnaga et al. 2001], each node keeps the number of elements reachable by the path starting from the root node to the node. As shown in Figure 7-(a), a node in our path tree keeps the number of subnodes including itself which we call the `weighted frequency`. Thus, intervals for higher level elements are enlarged and the intervals for simple paths do not shrink quickly. However, as mentioned in [Aboulnaga et al. 2001], the path tree consumes a large amount of memory, in the worst case, \(O(E)\), where \(E\) is the number of elements.

Thus, instead of using the path tree, we use a simple heuristic: if we visit an element whose tag is a new tag, then we increase the frequencies of elements which are ancestors of the currently visited element. Thus, like the path tree, the intervals for higher level elements are enlarged. We call this frequency the `adjusted frequency`. 

![Fig. 7. Various frequencies](image-url)

**ACM Transactions on Internet Technology, Vol. V, No. N, Month 20YY.**
Our simple heuristic method requires $O(L)$ space, where $L$ is the length of the longest simple path in given XML data. Furthermore, our method is more efficient than that of the path tree. Whenever a new node in the path tree is created, the weighted frequencies of ancestor nodes of the new node should be increased by 1. However, our method increases the adjusted frequencies of ancestor nodes when an element with a new tag appears. As illustrated in Figure 7-(b), with the reduction of space requirement and enhanced performance, we can obtain the statistics similar to those of the path tree.

Procedure Statistics_Collection(Token, Pathstack, Elemhash)
begin
1. if (Token is START-TAG) {
2. Pathstack.push(Token)
3. eleminfo := Elemhash.hash(Token)
4. if (eleminfo = NULL) {
5. eleminfo := new ELEMINFO(Token)
6. Elemhash.insert(eleminfo)
7. for each token $t$ in Pathstack do {
8. tempinfo := Elemhash.hash($t$)
9. tempinfo.adjusted_frequency += 1
10. Elemhash.total_frequency += 1
11. }
12. }
13. } else // Token is END-TAG
14. Pathstack.pop()
end

Fig. 8. The algorithm of the statistics collector

The algorithm of the statistics collector is presented in Figure 8. The input token is a tag. The trace of the currently visited element is kept by Pathstack (Line (2) and Line (14)). The hash at Line (3) is the hash function which returns an ELEMINFO for a given tag. Thus, when an element with a new tag appears, the hash function returns NULL (Line (4)). Then, the statistics collector makes an ELEMINFO for the element (Line (5)-(6)) and increases the adjusted frequencies for ancestors of the element including itself (Line (7)-(11)). At Line (10), we accumulate the total frequency to normalize the adjusted frequencies.

To produce the statistics of the inferred type for data values of each distinct element, the ELEMINFO has six fields: inferred_type, min, max, symhash, chars_frequency and Tag. The inferred_type field keeps the type of data values, up to now. The inferred_type is set as undefined initially. The min and max fields keep the track of the minimum binary value and the maximum binary value of data values, respectively. The symhash field is a hash table which keeps distinct data values. This symhash can be used as a dictionary for the dictionary encoder when the type of an element is the enumeration. The chars_frequency is an integer array which keeps the frequencies of individual characters of data values. This chars_frequency field is used to build a huffman tree for the huffman encoder. The Tag field is used to keep the name of the element. To obtain the proper statistics of data values of each distinct element, the algorithm of the type inference engine shown in Figure 9 is executed.

The input token of Type_Inferencing is a data value. As mentioned above, Pathstack keeps the trace of currently visited elements. Thus, the tag of the element which is the owner of the given data value is at the top of Pathstack (Line (1)
Procedure TypeInferencing(Token, Pathstack, Elemhash)
begin
1. Tag := Pathstack.top()
2. eleminfo := Elemhash.hash(Tag)
3. type := Infer_Type(Token)
4. switch(eleminfo.inferred_type) {
5. case undefined :
6. case integer :
7. if(type = integer) {
8. eleminfo.inferred_type := integer
9. intvalue := get_IntValue(Token)
10. eleminfo.min := MIN(eleminfo.min, intvalue)
11. eleminfo.max := MAX(eleminfo.max, intvalue)
12. eleminfo.symhash.insert(Token)
13. eleminfo.accumulate_chars_fq(Token)
14. }
15. else if(type = float) {
16. eleminfo.inferred_type := float
17. floatvalue := get_FloatValue(Token)
18. eleminfo.min := MIN(eleminfo.min, floatvalue)
19. eleminfo.max := MAX(eleminfo.max, floatvalue)
20. eleminfo.symhash.insert(Token)
21. eleminfo.accumulate_chars_fq(Token)
22. }
23. else { // string
24. eleminfo.symhash.insert(Token)
25. if(the number of entries in eleminfo.symhash < 128) {
26. eleminfo.inferred_type := enumeration
27. } else eleminfo.inferred_type := string
28. eleminfo.accumulate_chars_fq(Token)
29. }
30. break
31. case float :
32. if(type = integer or type = float) {
33. eleminfo.inferred_type := float
34. floatvalue := get_FloatValue(Token)
35. eleminfo.min := MIN(eleminfo.min, floatvalue)
36. eleminfo.max := MAX(eleminfo.max, floatvalue)
37. eleminfo.symhash.insert(Token)
38. eleminfo.accumulate_chars_fq(Token)
39. }
40. else { // string
41. eleminfo.symhash.insert(Token)
42. if(the number of entries in eleminfo.symhash < 128) {
43. eleminfo.inferred_type := enumeration
44. } else eleminfo.inferred_type := string
45. eleminfo.accumulate_chars_fq(Token)
46. }
47. break
48. case enumeration :
49. ...
50. break
51. case string :
52. eleminfo.accumulate_chars_fq(Token)
53. break
54. }
end

Fig. 9. The algorithm of the type inference engine

in Figure 9). Therefore, we obtain the corresponding ELEMINFO using this tag easily (Line (2)).

The function Infer_Type at Line (3) infers the type of the given data value using a simple rule as follows:

If all characters of the data value are numeric (‘0’∼‘9’) and the first character
is not ‘0’, then Infer_Type returns *integer* which denotes that the data value is an integer.

If all characters of the data value are numeric (‘0’∼‘9’) with only one ‘.’ and the first and second characters are ‘0’ and ‘.’ (i.e., in case of 0dddd), respectively, or the first character and the last character are not ‘0’ nor ‘.’, respectively (i.e., in case of ddd.dddd), then a *float* is returned by Infer_Type. Otherwise, we consider the type of the data value as a *string*.

If the type of the element is an *integer* or *undefined* and the type of the given data value is an *integer* (Line (5)-(14)), then we transform the data value into a binary value (Line (9)) and adjust the *min* and *max* fields using the binary value (Line (10)-(11)). The inferred type can be changed even though the currently inferred type is an *integer*. Thus, to prepare for the future change, we also maintain the *symhash* field and *chars_frequency* field, properly (Line (12)-(13)).

If the type of the data value is a *float* (Line (15)-(22)), we change the *inferred_type* to be the *float* since the *integer* type does not express floats fully, whereas the *float* type can express integers. Then, similar to the case of *integer* just described, the data value is transformed into a binary value (Line (17)) and the *min* and *max* fields are adjusted (Line (18)-(19)). Also, the *symhash* and *chars_frequency* fields are maintained for the same reason (Line (20)-(21)).

If the type of the data value is a *string* (Line (23)-(29)), we change the *inferred_type*. Even though the preceding data values are integers, we change the *inferred_type* since the *integer* type does not express the string but the *string* type can express the numeric typed data using numeric characters. XPRESS has two types for textual data: *enumeration* and *string*. The *string* type is for general textual data, while the *enumeration* type is for the special string whose number of distinct values is less than 128. To keep the distinct values, the hash table, *symhash*, discards duplicated strings (Line (12), (20), and (24)). Thus, if the number of distinct entries of *symhash* is less than 128, we assign *enumeration* to the *inferred_type*. Otherwise, we assign *string* to the *inferred_type*. Also, because the *inferred_type* can be changed to *string*, the *chars_frequency* field is updated (Line (28)).

When the *inferred_type* is *float* (Line (31)-(47)), we perform the similar actions as we do in case that the *inferred_type* is *integer*. However, we manage the cases of *integer* and *float* together since the preceding data values are floats which can express all the integers. Whether the type of the data value is *integer* or *float*, we perform the exactly same actions we did through Line (16)-(21) (Line (33)-(38)). And, if the type of the data value is a *string* (Line (40)-(46)), we manage the exactly same actions we did through Line (24)-(28) (Line (41)-(45)).

However, when the *inferred_type* is *enumeration* (Line (48)-(50)), we only check whether the *inferred_type* can be changed to *string* without considering the type of the given data value. Thus, Line (49) is the same as Line (24)-(28). If the *inferred_type* is *string*, the *chars_frequency* field is updated only (Line (51)-(53)).

### 4.2 XML Encoder

In this section, we describe the details of XML Encoder which compresses XML data using various encoders.

There are six encoders for data values in XPRESS, shown in Table I. Each
distinct element has its own encoder which is one of six encoders.

<table>
<thead>
<tr>
<th>Encoder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u8</td>
<td>encoder for integers where max-min $&lt; 2^7$</td>
</tr>
<tr>
<td>u16</td>
<td>encoder for integers where $2^7 \leq \text{max-min} &lt; 2^{15}$</td>
</tr>
<tr>
<td>u32</td>
<td>encoder for integers where $2^{15} \leq \text{max-min} &lt; 2^{31}$</td>
</tr>
<tr>
<td>f32</td>
<td>encoder for floating values</td>
</tr>
<tr>
<td>dict8</td>
<td>dictionary encoder for enumeration typed data</td>
</tr>
<tr>
<td>huff</td>
<td>huffman encoder of textual data</td>
</tr>
</tbody>
</table>

Table I. Data Encoders

u8, u16, u32 and f32 are the differential encoders for numeric data and dict8 and huff are the encoders for textual data.

As mentioned in Section 3, the encoders for numeric data transform the numeric data into binary and apply differential encoding with the minimum value obtained by the type inference engine. Note that the most significant bit (MSB) of the encoded value by the numeric data encoders is 0. u8, u16 and u32 use 7 bits, 15 bits and 31 bits, and generate one byte, two bytes and four bytes, respectively.

A floating value generated by the encoder f32 is always positive since f32 generates the difference from the minimum value. Thus, the sign bit in Figure 10 is always 0. Also, the encoder dict8 uses maximally 7 bits since, as described in Section 4.1, the number of distinct string values is less than $128 = 2^7$. Thus, the MSB of one byte generated by dict8 is also 0.

In contrast to the other encoders, the encoder huff generates variable length encoded sequences. To parse this encoded sequence easily, we divide the encoded sequence into subsequences whose lengths are less than 128 and put one byte in front of each subsequence to denote the length of it. The encoded sequence whose length is less than 128 is not partitioned but has one byte for the length. Therefore, the MSB of each sequence or subsequence is always 0 since its length is less than 128. Consequently, in XPRESS, every MSB of encoded values for data values is 0.

Until now, we described the encoders for data values. Next, we present the encoder for tags.

Start tags of individual elements are encoded by reverse arithmetic encoding using simple paths. In practice, we implement an approximated encoder, called the approximated reverse arithmetic encoder (ARAE), to improve the compression ratio and to parse compressed XML data without ambiguity.

Every MSB of the code generated by ARAE is 1. As mentioned above, every MSB of encoded data values is 0. Thus, the parser for compressed XML data easily distinguishes data from structure.

To do this, ARAE adds 1.0 to the minimum floating value of the interval for a simple path. Since the minimum floating value generated by reverse arithmetic encoding is always positive, the sign bit of each MSB is 0. Consequently, every MSB of encoded values for data values is 0.

Fig. 10. IEEE 32 bit floating point standard 754

A floating value generated by the encoder f32 is always positive since f32 generates the difference from the minimum value. Thus, the sign bit in Figure 10 is always 0. Also, the encoder dict8 uses maximally 7 bits since, as described in Section 4.1, the number of distinct string values is less than $128 = 2^7$. Thus, the MSB of one byte generated by dict8 is also 0.

In contrast to the other encoders, the encoder huff generates variable length encoded sequences. To parse this encoded sequence easily, we divide the encoded sequence into subsequences whose lengths are less than 128 and put one byte in front of each subsequence to denote the length of it. The encoded sequence whose length is less than 128 is not partitioned but has one byte for the length. Therefore, the MSB of each sequence or subsequence is always 0 since its length is less than 128. Consequently, in XPRESS, every MSB of encoded values for data values is 0.

Until now, we described the encoders for data values. Next, we present the encoder for tags.

Start tags of individual elements are encoded by reverse arithmetic encoding using simple paths. In practice, we implement an approximated encoder, called the approximated reverse arithmetic encoder (ARAE), to improve the compression ratio and to parse compressed XML data without ambiguity.

Every MSB of the code generated by ARAE is 1. As mentioned above, every MSB of encoded data values is 0. Thus, the parser for compressed XML data easily distinguishes data from structure.

To do this, ARAE adds 1.0 to the minimum floating value of the interval for a simple path. Since the minimum floating value generated by reverse arithmetic encoding is always positive, the sign bit of each MSB is 0. Consequently, every MSB of encoded values for data values is 0.
encoding is in \([0.0, 1.0]\), the added value is in \([1.0, 2.0]\). According to the IEEE floating point representation (see Figure 10), a floating value is represented as 
\[ S \times 1.0 \times 2^{E-127}. \]
For example, the binary representation of 1.25 is 1.01 and this is transformed into \(1 \times 1.01 \times 2^0\). Thus, \(S = 0, E = 0+127 = 0111\ 1111,\) and \(M = [1.]01\). The first bit\(^2\) of the second byte for every floating value in \([1.0, 2.0]\) is always 1 since the sign bit and the biased exponent are 0 and 127 (= 0111 1111), respectively. Thus, by cutting the first byte, the MSB of the code generated by ARAE is always 1.

In addition, to reduce the size of compressed XML data, ARAE truncates the last byte. Due to the reduction of the precision, the code generated by ARAE may not always represent the corresponding simple path exactly. However, at least, the code generated by ARAE represents the tag of an element. As described in Example 3, the generated code still represents a label path (i.e., a suffix of a simple path).

**Example 3.** Suppose that ARAE truncates digits less than \(10^{-2}\) (i.e., last 17 bits) and that tags and corresponding Interval\(T\)s are the same as those in Example 1.

The interval for a simple path book.section, subsection.subtitle is \([1.0 + 0.9 + 0.1 \times 0.69 = 1.969, 1.0 + 0.9 + 0.1 \times 0.699 = 1.9699]\). Then, the truncated value is 1.96 which is in the interval \([1.96, 1.99]\) for subsection.subtitle.

Therefore, this approximation does not damage the accuracy and the efficiency of query processing. Recall that the reduction of data size by the data compression induces the performance improvement due to the reduction of disk I/Os. Furthermore, common structural constraints of XML queries are partial matching path expressions based on label paths instead of simple paths since users may not know or may not be concerned with the detailed structure of XML data and intentionally make the partial matching path expression to get intended results. But, note that too much approximation incurs the inefficiency of query processing since a label path represented by the encoded value becomes too short.

Finally, to distinguish start tags and end tags, the interval \([1.0+0.0 = 0x8000, 1.0+2^{-7} = 0x8100]\) is reserved. For all end tags, one byte 0x80 (= 1000 0000) is assigned since the codes for the interval start with 0x80. And codes for start tags are always greater than or equal to 0x8100. Therefore, the parser for compressed XML data distinguishes the codes for start tags and the codes for end tags.

The algorithm of XML Encoder is in Figure 11. First, XMLParser is reinitialized to rescan a given XML file (Line (1)). Then, for each distinct element, XML Encoder calculates Interval\(T\) and chooses a proper encoding method (e.g., u8) using the function Initialization (Line (2)). To compute Interval\(T\), we used the interval \([2^{-7}, 1.0-2^{-15}]\) as the entire interval instead of \([0.0, 1.0]\) since \([0.0, 2^{-7}]\) is reserved.

\(^2\)It is represented by the gray box in Figure 10
Procedure XMLEncoder(Elemhash)
begin
1. XMLParser.reinit()
2. Initialization(Elemhash)
3. Pathstack := new Stack()
4. Intervalstack := new Stack()
5. do {
6. Token := XMLParser.getToken()
7. if (Token is a tag)
8. ARAE(Token, Pathstack, Intervalstack, Elemhash)
9. else //Token is a data value
10. Encoding(Token, Pathstack, Elemhash)
11. } while (Token != EOF)
end

Fig. 11. The algorithm of XML Encoder

for end tags and the value less than $2^{-15}$ can not be represented using 15 bits. Also, for the same reason, we adjusted the length of $\text{Interval}_T$ to a number greater than $2^{-15}$. In general, this case does not appear.

Pathstack is used to keep the information of an owner element of data values (Line (3)). To compute the interval for the currently visited element, the interval for the parent element is required. To keep the interval for a parent element, a stack, called Intervalstack, is created (Line (4)). And then, the token generated by XMLParser is compressed by encoders of XPRESS (Line (5)-(11)).

4.3 Query Processing

To evaluate queries on compressed XML data generated by XPRESS, we devise a query processor. The query processor consists of a query parser, a query transformer, and a query executor.

The query parser separates the values from the label path expression when the query contains any value comparison. Also, to support complicated partial matching path queries whose examples are provided in Table III of Section 6.1, the query parser breaks down a complicated path expression into multiple single path expressions\textsuperscript{3} according to the occurrence of `/`.

For example, a complicated path expression $P = //p_1/p_2/p_3//p_4/p_5/p_6$ is partitioned into $P_1$ and $P_2$, where $P_1 = //p_1/p_2/p_3$ and $P_2 = //p_4/p_5/p_6$. The query executor looks for the elements which satisfy $P_1$ and evaluates $P_2$ among the descendants of $P_1$’s results.

The query transformer transforms the single path expressions to intervals. First, the query transformer partitions each long single path expression obtained by the query parser into short single path expressions whose corresponding interval sizes are greater than $1.0 + 2^{-15}$ since an interval for each element is expressed based on the precision of $1.0 + 2^{-15}$, as mentioned in Section 4.2. Suppose that a single path expression $P_s = //p_1/\ldots/p_n$ requires a precision higher than $1.0 + 2^{-15}$ and $P_v = //p_1/\ldots/p_i$ requires a precision lower than $1.0 + 2^{-15}$. Then, $P_s$ is partitioned as $P_s = P_s'P_s''$, where $P_s'' = /p_{i+1}/\ldots/p_n$. Also, if a precision higher than $1.0+2^{-15}$ is required for $P_s''$, we apply this partitioning process repeatedly. Thus, the query

\textsuperscript{3}A single path expression denotes a path expression that contains at most one occurrence of `/`.  

ACM Transactions on Internet Technology, Vol. V, No. N, Month 20YY.
transformer transforms the partitioned single path expression into a sequence of intervals. However, the length of the interval sequence is 1 since a single path expression is generally short.

Finally, by using the sequence of intervals transformed by the query transformer, the query executor evaluates the tokens of encoded elements in compressed XML data whether their encoded values are in an interval of the sequence or not.

For the exact matching query, a data value of exact matching conditions in a query is converted into an encoded value using the type dependent encoder as described in Section 4.2. Then, the query executor detects the elements which satisfy the label path expression and the value comparison without decompression. For the range query, the range condition for a numeric typed element is encoded by the type dependent encoder for the element. Then, without the decompression of encoded values, the query is evaluated since the type dependent encoders for numeric typed elements preserve the order information among data values. However, for the range condition of a textual typed element, a partial decompression is required since our encoders (i.e., huff and dict8) for textual data do not preserve the order information among data values. Therefore, to obtain the data values that fall within the range condition, the decompression of encoded data values is required.

5. UPDATE PROCESSING

In this section, we present the details of the update processor which supports direct updates on compressed XML data.

The naive approach for updates on compressed XML data is that original XML data is constructed by the complete decompression and the update and the recompression are performed on uncompressed XML data. In this case, the system resource and time are wasted. Thus, we devised an update processor for compressed XML data which does not perform a complete decompression.

Basically, the update operations can be categorized by three types: deletion, insertion, and change. The deletion is trivially performed by eliminating a certain portion of compressed XML data specified by the query. The change is considered as the combination of the deletion and the insertion. Thus, in this paper, we only focus on the insertion on compressed XML data.

As mentioned earlier, XPRESS encodes tags and data values according to the statistics (e.g., frequencies of tags, types of data values). However, the newly inserted XML fragment may affect currently compressed XML data.

Thus, by analyzing the newly inserted XML fragment, the update processor of XPRESS renews the statistics. And then, with respect to the difference between the renewed statistics and the current statistics, a partial decompression of compressed XML data is performed. Using the renewed statistics, the XML fragment and the partially decompressed XML data are recompressed.

The main algorithm for the update processor of XPRESS is depicted in Figure 12. Two input parameters of XMLUpdater are Xpath and XMLFragment. Xpath is an XPath query which is used to specify the insertion points on the compressed XML data. And, XMLFragment is the newly inserted XML fragment.

First, to parse XML fragment, XMLParser is initialized with the given XML fragment (Line (1)). At Line (2)-(5), the algorithm XMLUpdater generates two
Procedure XMLUpdater(Xpath, XMLFragment)
begin
1. XMLParser.init(XMLFragment)
2. OLD_Elemhash := new Hash()
3. OLD_Elemhash.resume()
4. NEW_Elemhash := new Hash()
5. NEW_Elemhash.resume()
6. UpdateAnalyzer(OLD_Elemhash, NEW_Elemhash)
7. UPDATEPOINT := get_updatepoint(Xpath)
8. Updating(UPDATEPOINT, OLD_Elemhash, NEW_Elemhash)
end

Fig. 12. The algorithm of XML Updater

hash tables: OLD_Elemhash and NEW_Elemhash. OLD_Elemhash is used to keep the current statistics and NEW_Elemhash is used to keep the renewed statistics. By invoking the method resume, the statistics are reloaded into each hash table from the header of compressed XML data. At the beginning, OLD_Elemhash and NEW_Elemhash are the same, and NEW_Elemhash is changed subsequently.

The core modules of XMLUpdater are Update Analyzer and Updating. Similar to the compression scheme of XPRESS that is categorized as the semi-adaptive compression, XMLUpdater of XPRESS requires two scans of XML fragment. The procedure UpdateAnalyzer (see details in Section 5.1) is invoked to renew the statistics (Line (6)).

By using the query processor of XPRESS, the get_updatepoint function at Line (7) finds an UPDATEPOINT where the XML fragment is required to be inserted. For brevity, in Figure 12, we assume that there is only one UPDATEPOINT on given compressed XML data. However, the extension to maintain multiple UPDATEPOINTs is straightforward.

The procedure Updating (see details in Section 5.2) is invoked to insert the XML fragment at the UPDATEPOINT and re-encode certain portions of compressed XML data using the statistics gathered by the procedure UpdateAnalyzer (Line (8)).

5.1 Update Analyzer

The algorithm of Update Analyzer is presented in Figure 13. Basically, Update Analyzer renews the statistics by scanning the XML fragment.

Pathstack is used to identify the tag of owner element of a data value in the XML fragment.

The newly inserted XML fragment may incur two kinds of violations with respect to the current statistics kept in OLD_Elemhash. The first is the appearance of new tags. The second is the change of the inferred data type.

As described in Section 3, the reverse arithmetic encoder for tags partitions the entire interval [0,1) into subintervals, one for each distinct tag. Thus, the intervals for new tags do not exist in the current statistics which are kept in OLD_Elemhash. Thus, to assign the intervals to new tags, the frequency of new tags is computed. When an element with a new tag appears (Line (7)-(15)), the hash function of OLD_Elemhash returns NULL since the tag has not appeared on compressed XML data before (Line (7)). Also, if this tag appears for the first time, the hash function of NEW_Elemhash returns NULL. In this case, the procedure UpdateAnalyzer
A Compressor for Effective Archiving, Retrieval, and Update of XML Documents

Procedure Update_Analyzer(OLD_Elemhash, NEW_Elemhash, Pathstack)
begin
    1. Pathstack := new stack()
    2. do
        3. Token := XMLParser.get_Token()
        4. if(Token is START_TAG) {
            5. Pathstack.push(Token)
            6. eleminfo := OLD_Elemhash.hash(Token)
            7. if(eleminfo = NULL) {
                8. eleminfo := NEW_Elemhash.hash(Token)
                9. if(eleminfo = NULL) {
                    10. eleminfo := new ELEMINFO(Token)
                    11. NEW_Elemhash.insert(eleminfo)
                }
            }
            12. eleminfo.adjusted_frequency += 1
            13. NEW_Elemhash.total_frequency += 1
        } else if(Token is END_TAG) {
            14. Pathstack.pop()
        } else //Token is a data value
            15. Type_Inferencing(Token, Pathstack, NEW_Elemhash)
        } while(Token != EOF)
end

Fig. 13. The algorithm of Update Analyzer

makes an ELEMINFO for the element and inserts it into NEW_Elemhash (Line (9)-(12)). Then, Update_Analyzer increases the adjusted frequency of the element and the total frequency of NEW_Elemhash by 1 (Line (13)-(14)).

Lastly, for data values in the XML fragment, the algorithm of the type inference engine described in Figure 9 is invoked with NEW_Elemhash (Line (19)). Thus, by the comparison between type information kept in OLD_Elemhash and type information kept in NEW_Elemhash, we can identify the change of the data type.

5.2 Updating

The Updating module inserts the XML fragment into UPDATEPOINT on compressed XML data and changes portions of compressed XML data which are affected by the XML fragment. The algorithm of Updating is shown in Figure 14.

At the beginning of the algorithm Updating, the initialization for NEW_Elemhash is performed to calculate Interval_T and choose a proper encoding method as described in Section 4.2 (Line (1)).

For the initialization of OLD_Elemhash, choosing proper decoding methods is additionally required since the partial decompression of compressed data values is necessary if the inferred_type is changed to another type (Line (2)).

NEW_Codestack is used to keep the trace of currently visiting element by using the encoded value for the tag (Line (3)). In this case, if new tags appear in the XML fragment, NEW_Codestack is maintained with the newly calculated intervals from NEW_Elemhash after invoking Update_Analyzer.

As mentioned previously, our proposed update processor directly inserts the XML fragment into compressed XML data. Thus, to parse compressed XML data, a specific parser called CompressedXMLParser is used. CompressedXMLParser generates a Comp_Token while traversing compressed XML data. As mentioned in Section 4, classifying Comp_Tokens as START_TAG, END_TAG, and data values is easy since all the MSBs of the encoded values for tags start with 1 while all the
Procedure Updating(UPDATEPOINT, OLD_Elemhash, NEW_Elemhash, XMLFragment)
begin
1. Initialization(NEW_Elemhash)
2. Initialization(OLD_Elemhash)
3. NEW_Codestack := new stack()
4. do
5. if(CompressedXMLParser.getPosition() = UPDATEPOINT) {
6. XMLParser.reinit(XMLFragment)
7. insert XMLFragment using NEW_Elemhash and NEW_Codestack
8. // Similar to XML Encoder in Figure 11
9. }
10. Comp_Token := CompressedXMLParser.get_Token()
11. if(Comp_Token is START_TAG) {
12. if(OLD_Elemhash.total_frequency != NEW_Elemhash.total_frequency) {
13. old_eleminfo := OLD_Elemhash.getInfo(Comp_Token)
14. Comp_Token := ARAE_for_Update(old_eleminfo.Tag, NEW_Codestack, NEW_Elemhash)
15. }
16. NEW_Codestack.push(Comp_Token)
17. } else if(Comp_Token is END_TAG) {
18. NEW_Codestack.pop()
19. }
20. else { // Comp_Token is a data value
21. NEW_Code := NEW_Codestack.top()
22. new_eleminfo := NEW_Elemhash.getInfo(NEW_Code)
23. old_eleminfo := OLD_Elemhash.hash(new_eleminfo.Tag)
24. if(old_eleminfo.inferred_type != new_eleminfo.inferred_type) {
25. value := Decoding(Comp_Token, old_eleminfo)
26. Encoding_for_Update(value, new_eleminfo)
27. }
28. }
29. } while(Comp_Token != EOF)
end

Fig. 14. The algorithm of Updating

MSBs for data values are 0. Especially, for END_TAGs, XPRESS assigned 0x80 (=1000 0000).

When the parsing position of CompressedXMLParser is the same as UPDATE-POINT, the XML fragment is inserted (Line (5)-(9)). First, XMLParser is reinitialized for the XML fragment (Line (6)). Next, as commented at Line (8), the XML fragment is compressed by using NEW_Elemhash and NEW_Codestack, and then inserted. This process is similar to XML Encoder described in Figure 11.

As described in Section 5.1, the total frequency of NEW_Elemhash is increased only when a new tag appears in the XML fragment. Thus, if the XML fragment has new tags, the total frequency of OLD_Elemhash and the total frequency of NEW_Elemhash are different.

For each START_TAG, if the two total frequencies are different, a new encoded value for each tag is required since the subinterval, Interval_T, for each tag is changed (Line (12)-(15)). Otherwise, Comp_Token is simply pushed into NEW_Codestack (Line (16)).

Note that the value of Comp_Token is the minimum value of the interval generated by the reverse arithmetic encoder using OLD_Elemhash. By checking the containment relationship between the minimum value (= Comp_Token) and each Interval_T in OLD_Elemhash, proper element information (i.e., ELEMINFO) can be obtained. Thus, the procedure Updating obtains the related ELEMINFO, old_eleminfo, by calling OLD_Elemhash.getInfo (Line (13)).

From old_eleminfo, the procedure Updating gets the tag (= old_eleminfo.Tag) of
currently visiting element and the new encoded value of the parent element’s tag which is kept at the top of NEW_Codestack. Thus, by invoking ARAE_for_Update, the new encoded value of Comp_Token is computed and replaced (Line (14)).

Lastly, for data values, the algorithm Updating obtains an ELEMIINFO from OLD_Elemhash, and an ELEMIINFO from NEW_Elemhash (Line (21)-(23)). The encoded value of the tag of the element which is the owner of the given compressed data value is at the top of NEW_Codestack. Thus, by calling NEW_Elemhash.get_info with the encoded value of the owner element’s tag, the new_eleminfo is obtained. And, using new_eleminfo.Tag, the old_eleminfo is acquired from OLD_Elemhash.

Then, their inferred_types are compared at Line (24). Thus, if their inferred_types are different, the decompression of the compressed data value (i.e., Comp_Token) and the recompression of the decompressed value are performed (Line (25)-(26)).

Note that, at Line (24), much more precise comparison is required since the min or max can be changed even though the inferred_type is not changed. In Figure 14, we did not show the comparisons for such cases for brevity. However, an extension to handle such cases is straightforward.

6. EXPERIMENTS

To show the effectiveness of XPRESS, we empirically compared the performance of XPRESS with two representative XML compressors XMill\(^4\) and XGrind\(^5\) as well as a general compressor, gzip, using real-life XML data sets. In our experiments, XPRESS shows a reasonable compression ratio compared to XMill. To the best of our knowledge, there is no XML compressor which supports querying compressed XML data except XGrind. Thus, we compared the query performance of XPRESS to that of XGrind. Also, since there is no such XML compressor which supports direct updates on compressed XML data, for the efficiency of the update processor of XPRESS, we compared the update performance of XPRESS with the naive approach. As mentioned in Section 5, the naive approach decompresses compressed XML data, updates decompressed XML data, and, finally, recompresses updated XML data. As a result, XPRESS shows significantly better query performance than XGrind and update performance than the naive approach.

6.1 Experimental Environment

The experiments are performed on a Sun Ultra Sparc II 168MHz platform with Solaris 2.5.1 and 384 MBytes of main memory. The data sets were stored on a local disk. In our experiments, XMill does not have any user-specified encoders. In XGrind, the query processor of compressed XML data does not support partial matching path queries. Thus, we implemented a query processor which supports partial matching path queries in XGrind.

Data Sets We evaluated XPRESS using three real-life XML data sets: Baseball, Course, and Hamlet. The characteristics of the data sets used in our experiment are summarized in Table II. Size denotes the disk space of XML data in MBytes, Depth is the length of the longest simple path of each XML data set, Tags indicate the number of distinct tags, Numeric represents the number of distinct elements

\(^4\)available in http://www.researech.att.com/sw/tools/xmill/
\(^5\)available in http://sourceforge.net/projects/xgrind/
whose data values’ type is numeric (i.e., integer or float), and Enum indicates the number of distinct elements whose data values’ type is enumeration.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Size</th>
<th>Depth</th>
<th>Tags</th>
<th>Numeric</th>
<th>Enum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseball</td>
<td>1.01</td>
<td>6</td>
<td>46</td>
<td>19</td>
<td>5</td>
</tr>
<tr>
<td>Course</td>
<td>2.91</td>
<td>6</td>
<td>18</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Hamlet</td>
<td>0.28</td>
<td>5</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table II. XML Data Set

The Baseball [Harold] contains the complete baseball statistics of all players of each team that participated in the 1998 Major League. Since it contains statistics, it has many integer and float typed values.

The Course [Anonymous] addresses the description of courses held in the University of Washington. Since it is for the description of courses, it has some integer values to indicate schedule lines, credits, and class rooms, and some enumerated values to describe course code, title, days of classes, and building names.

The Hamlet [Cover 2001] is one of the most famous plays of Shakespeare. Since it describes the overall scenario of the play, Hamlet, this XML document does not have any numeric typed elements nor enumeration typed elements.

However, to test the effectiveness of XPRESS on large sized XML data, we scaled up each original data by 1, 10, 50, and 100 times, respectively.

<table>
<thead>
<tr>
<th>Name</th>
<th>Query Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>/SEASON/LEAGUE/DIVISION/TEAM/PLAYER/GIVEN_NAME</td>
</tr>
<tr>
<td>B2</td>
<td>//TEAM/PLAYER/SURNAME</td>
</tr>
<tr>
<td>B3</td>
<td>//LEAGUE/DIVISION//PLAYER/WINS</td>
</tr>
<tr>
<td>B4</td>
<td>//LEAGUE/DIVISION//PLAYER[WINS &gt;= 0 and WINS &lt;= 10]/WINS</td>
</tr>
<tr>
<td>C1</td>
<td>/root/course/section/session/place/building</td>
</tr>
<tr>
<td>C2</td>
<td>//session/time</td>
</tr>
<tr>
<td>C3</td>
<td>//course/section//place/building</td>
</tr>
<tr>
<td>C4</td>
<td>//course/section//place[building &gt; KNE and building &lt; SMI]/building</td>
</tr>
<tr>
<td>H1</td>
<td>/PLAY/ACT/SCENE/SPEECH/SPEAKER</td>
</tr>
<tr>
<td>H2</td>
<td>//PGROUP/PERSONA</td>
</tr>
<tr>
<td>H3</td>
<td>//ACT/SCENE//SPEAKER</td>
</tr>
<tr>
<td>H4</td>
<td>//ACT/SCENE//SPEAKER[text() &gt;= BERNARDO and text() &lt;= HAMLET]</td>
</tr>
</tbody>
</table>

Table III. XML Query Set

**Queries** We evaluated XPRESS using several queries. The characteristics of queries used in our experiment are described in Table III.

The first character in the first column indicates the data set on which the query is executed: ‘H’ denotes the Hamlet, ‘B’ is for the Baseball and ‘C’ is for the Course. The number in the first column represents the type of queries. The queries of type 1 are path expressions based on the simple path, the queries of type 2 are partial matching path expressions, the queries of type 3 are complicated partial matching path expressions, and the queries of type 4 are complicated partial matching path queries with the range of data values. Query Definition in Table III describes the corresponding XQuery queries.

We choose these kinds queries for the following reasons. Queries of type 1 evaluate the query performance for long path expressions. Queries of type 2 test the query performance of simple partial matching path queries. Query type 3 is similar to query type 2 but is more complicated than query type 2. Finally, to measure the...
query performance of range queries, we choose the query type 4. The query type 4 represents a general query style since, generally, users do not know the entire structure of XML data and want to select XML fragmentations for a certain range of values. In addition, B4 is the range query of the integer typed data values, C4 is the range query of the enumeration typed data values and H4 is the range query of the general textual data values.

<table>
<thead>
<tr>
<th>Name</th>
<th>Update Position</th>
<th>XML Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>UB1</td>
<td>//DIVISION/TEAM/PLAYER</td>
<td>&lt;WINS&gt;3&lt;/WINS&gt;</td>
</tr>
<tr>
<td>UB2</td>
<td>//DIVISION/TEAM/PLAYER</td>
<td>&lt;MVP&gt;0&lt;/MVP&gt;</td>
</tr>
<tr>
<td>UB3</td>
<td>//DIVISION/TEAM/PLAYER</td>
<td>&lt;WINS&gt;3.2&lt;/WINS&gt;</td>
</tr>
<tr>
<td>UC1</td>
<td>//section/session/time</td>
<td>&lt;start_time&gt;940&lt;/start_time&gt;</td>
</tr>
<tr>
<td>UC2</td>
<td>//section/session/time</td>
<td>&lt;break_time&gt;10&lt;/break_time&gt;</td>
</tr>
<tr>
<td>UC3</td>
<td>//section/session/time</td>
<td>&lt;start_time&gt;ten&lt;/start_time&gt;</td>
</tr>
<tr>
<td>UH1</td>
<td>//ACT/SCENE/SPEECH</td>
<td>&lt;STAGEDIR&gt;MACBETH&lt;/STAGEDIR&gt;</td>
</tr>
<tr>
<td>UH2</td>
<td>//ACT/SCENE/SPEECH</td>
<td>&lt;SPEECH_TIME&gt;10&lt;/SPEECH_TIME&gt;</td>
</tr>
<tr>
<td>UH3</td>
<td>//ACT/SCENE/SPEECH</td>
<td>&lt;SPEECH_TIME&gt;ten&lt;/SPEECH_TIME&gt;</td>
</tr>
</tbody>
</table>

Table IV. XML Update Query Set

**Update Queries** We evaluated the update processor of XPRESS using several updates. The characteristics of updates are described in Table IV.

The first character ‘U’ in the first column indicates that the queries are for updates on compressed XML data. The second character and the number are used to denote the data set and the type, respectively. In Table IV, the updates of type 1 are the insertions of already appeared tags, the updates of type 2 are the insertions of newly appeared tags, and the updates of type 3 are the insertions of already appeared tags while the types of data values are changed from the integer to the float for UB3 and from the integer to the enumeration for UC3 and UH3. Update Position in Table IV describes the positions where the insertions are performed, based on the XPath queries. XML Fragment shows the newly inserting XML fragment.

### 6.2 Experimental Results

In this section, we first present the compression ratio of each compressor. The compression ratio is defined as follows:

\[
\text{Compression ratio} = 1 - \frac{\text{Size of compressed XML data}}{\text{Size of original XML data}}
\]

And, we report the compression time of each compressor. In addition, to show the effect of zlib in XMILL, we show the compression ratio of gzip which is applied to the compressed XML data of XPRESS and XGrind. Then, we show the query performance of XPRESS with that of XGrind. And, the update performance of XPRESS with that of the naive approach is presented at last.

Figure 15 shows the compression ratios for different data sets and compressors. As mentioned in Section 6.1, each XML data set is increased by 1, 10, 50, and 100 times to test XPRESS on large sized XML data. For each different size of XML data set, the four connected bars represent XMILL, gzip, XGrind, and XPRESS. Since XMILL uses the dictionary encoding method for structural information, and groups semantically related data values into containers before compressing with...
Fig. 15. Compression ratio

zlib, as we expected, XMill achieved the best compression ratio, on the average of 90%. The average compression ratio of XPRESS is 75%. Since XPRESS uses the type inference engine to apply appropriate compression methods for data values, it performs well if the data values are enumeration, floating, or integer type. Thus, the compression ratio of XPRESS for the Baseball is better than that for the other data sets since the Baseball contains many numeric typed data values. As shown in Table II, the Hamlet does not contain any numeric and enumeration typed data. For the Hamlet data set, the compression ratio of XPRESS is just slightly higher than that of XGrind since the huffman encoder in XPRESS inserts the length of an encoded value in front of the value. As a result, XPRESS shows a reasonable compression ratio for all cases.

Fig. 16. Compression time (log scale)

Figure 16 shows the compression time of each compressor based on the log scale. In our experiments, XGrind shows the worst compression time. As mentioned earlier, to determine the data value encoders (i.e., huffman encoding and dictionary encoding), XGrind uses DTDs. To parse and obtain some information from DTDs, XGrind adopts a shareware XML parser. Thus, the overhead of XML parsing and

\footnote{some bars for compression time of x1s are not shown in the graph since their values are less than or equal to 1.}
DTD validation is huge. In contrast to XGrind, XPRESS and XMill parse the XML document efficiently since they do not use any information from DTDs.

Also, XGrind encodes data values using the huffman encoder. Generally, huffman encoding is less efficient than differential encoding due to the massive traverse of the huffman tree. Furthermore, XGrind checks whether encoded data values by huffman encoding have predefined symbols for tags and inserts escaped characters to parse compressed XML data easily. However, the huffman encoder in XPRESS does not use the same procedure since it locates the length of an encoded value in front of the value. In addition, using proper encoding methods that are determined by the inferred types, XPRESS has much better compression time compared to that of XGrind. XMill and gzip show the best performance of data compression since they compress XML data by one scan. In the evaluation of the decompression time, the result shows the similar pattern compared to that of the compression time. Thus, we omit the graph of the decompression time.

In addition, to show the effect of the built-in compression library zlib in XMill, we re-compressed the compressed files generated by XGrind and XPRESS using gzip which uses zlib internally. The result is shown in Figure 17. In Figure 17, as we expected, XMill still shows the best compression ratio. Since XMill groups semantically related data values into same containers, zlib effectively compresses XML data. However, the compression ratios of the re-compressed XML data by gzip are very close to that of XMill. Thus, for archiving, applying gzip selectively for compressed XML data which is seldom queried is another alternative.

Although XMill shows the best performance in the compression ratio and the compression time, XMill does not support querying compressed XML data. Thus, to show the effectiveness of XPRESS, we compared the query performance of XPRESS to that of XGrind which support querying compressed XML data.

We plotted the query processing cost of all the queries for the three data sets based on the increased sizes of data sets in Figure 18. Basically, Figure 18 (a) represents the original sizes of the data sets provided in Table II, (b) represents the increased data sets by 10 times, (c) is by 50 times, and (d) is by 100 times, respectively. Also, for each query, the two connected bars represent XGrind and XPRESS. As shown in Figure 18, the query performance of XPRESS outperforms that of XGrind over all cases.
Fig. 18. Query evaluation time

The query cost of query type 1 (i.e., B1, C1, and H1) shows that the approximated reverse arithmetic encoder does not incur the degradation of efficiency. Since the lengths of path expressions in query type 2 (B2, C2, and H2) are short, the query processing cost is cheap. Thus, the difference of query performance between XPRESS and XGrind is not conspicuous. However, the query performance of XPRESS for complicated path expressions (B3, C3, and H3) outperforms that of XGrind since the query processor of XPRESS efficiently evaluates the queries using reverse arithmetic encoding. Also, for range queries, the performance gap increases since XPRESS minimizes the overhead of a partial decompression using order preserved encoders. Of particular interest is the performance gap for H4. The huffman encoder of XPRESS inserts the length of an encoded sequence in front of the sequence, while the huffman encoder of XGrind inserts escaped characters into an encoded sequence. Thus, in XGrind, escaped characters are eliminated from the encoded sequences to decompress the encoded sequence by the huffman decoder. Therefore, the overhead of a partial decompression of XPRESS is less than that of XGrind. On the average, the query performance of XPRESS is 2.99 times better than that of XGrind.

Lastly, we demonstrate the efficiency of the update processor of XPRESS. As mentioned previously, we compared the update performance of XPRESS to that of a naive approach which decompresses compressed XML data, makes updates, and recompresses updated XML data.

We provided the processing cost of the updates in Table IV for the three data sets, especially each data set is scaled up by 50 times. In Figure 19, (a) represents the update type 1 provided in Table IV, (b) represents the update type 2, and (c) is
for the update type 3, respectively. The two connected bars in Figure 19 represent the naive approach and XPRESS, respectively. As we expected, the update performance of XPRESS outperforms that of the naive approach over all cases since the update processor of XPRESS does not perform the complete decompression of compressed XML data.

Figure 19-(a) shows that the update processor of XPRESS efficiently inserts the tags already appeared on compressed XML data since the new encoded values for all the tags are not required as described in Figure 14. However, for update type 2, due to the insertion of the newly appeared tags, the recompression for the new encoded values for all the tags is required. Also, for update type 3, since the type of data values is changed as shown in Table IV, a partial decompression of the data values and the reencoding of data values using the newly assigned encoder are performed. Therefore, the updates of type 2 and 3 consume more time than those of type 1. However, the update performance of XPRESS outperforms that of the naive approach since the updater of XPRESS performs only a partial decompression, instead of the complete decompression. As a result, the performance of the update processor of XPRESS is about 8.78 times faster than that of the naive approach.

Consequently, XPRESS achieves significantly improved query performance compared to XGrind and shows the reasonable compression ratio. Also, XPRESS achieves significantly improved update performance compared to the naive approach.
7. CONCLUSION

In this paper, we propose XPRESS, an XML compressor which supports direct updates and efficient querying on compressed XML data. In XPRESS, we devise a novel encoding method, called reverse arithmetic encoding, which encodes a label path to a distinct interval in $[0.0, 1.0)$. Using the containment relationships among the intervals, path expressions are evaluated on compressed XML data effectively. Furthermore, to save the disk space, we implement the approximated reverse arithmetic encoder which does not incur the loss of the accuracy and the efficiency. Also, to apply proper encoders for data values, we devise an efficient type inference engine and, by inferred type information, XPRESS encodes the data values. Since the encoders for numeric typed data values do not lose the order information, we reduce the overhead of a partial decompression for range queries.

We implemented XPRESS, a query processor and an update processor for compressed XML data. To show the efficiency of XPRESS, we conducted an extensive experimental study with real-life XML data sets. Experimental results show that XPRESS improves query performance significantly. The compression ratio of XPRESS is superior to that of another XML compressor which supports direct querying compressed XML data. On the average, the query performance of XPRESS is 2.99 times better than that of an existing XML compressor and the compression ratio of XPRESS is 75%. Also, we demonstrated the efficiency of the update performance of XPRESS by comparing with that of a naive approach.

Currently, the type inference engine of XPRESS distinguishes the numeric data and textual data. Thus, for our future work, we plan to extent XPRESS to support complex typed data values such as URI (Uniform Resource Identifier) using data mining algorithms.

REFERENCES


ACM Transactions on Internet Technology, Vol. V, No. N, Month 20YY.


