XPRESS: A Queriable Compression for XML Data

Jun-Ki Min Myung-Jae Park Chin-Wan Chung

Division of Computer Science
Department of Electrical Engineering & Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
Taejon, Korea
{jkmin, jpark, chungcw}@islab.kaist.ac.kr

ABSTRACT

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. Thus, the query performance on compressed XML data is degraded.

In this paper, we propose XPRESS, an XML compressor which supports direct 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 2.83 times better than that of an existing XML compressor and the compression ratio of XPRESS is 73%.

1. INTRODUCTION

The eXtensible Markup Language (XML) [4] 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. 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. XML data is irregularly structured and self-describing like semistructured data. Using tags, XML separates contents and the representation (i.e., structure) in XML documents.

To retrieve XML data, XML query languages such as XPath [6] and XQuery [3] 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 [11, 12].

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.

Currently, a variety of research in the XML area has focused on issues related to XML storage [10], retrieval [8, 17], path indexes [5, 11]and publication [9, 19]. 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 is required.

One of the interesting applications for file based XML 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 [16, 23].

XMill [16] was designed to minimize the size of compressed XML data. However, XMill was not intended to support querying compressed XML data.

Recently, XGrind [23] 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. Tags and data values are encoded by dictionary encoding and huffman encoding [15]. To evaluate a path expression, the query processor parses and traverses compressed XML data. And, whenever a new element (or attribute) is visited by the query processor, the query processor finds the simple path of the visited element (or attribute) and checks whether the incoming path satisfies the given path expression. Furthermore, some kinds of queries such as range queries always require

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

the partial decompression of compressed XML data.

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 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 "$\exists$(// 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.

XPRESS has the following novel combination of characteristics to compress and retrieve 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., huffuman 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.

- **Semi-adaptive Approach:** Our compression scheme is categorized as the semi-adaptive approach [14] 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 allow 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. On the average, the query performance of XPRESS is 2.83 times better than that of an existing XML compressor and the compression ratio of XPRESS is 73%.

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 contains the result of our experiments which compares the performance of XPRESS to those of the other XML compressors. Finally, in Section 6, 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 [15, 20].

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 database systems is enhanced.

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 [14]. 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. In the semi-adaptive compression, the statistics are not changed during the compression phase. The adaptive compression does not require any prior statistics. 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. Encoded values by the dictionary encoding method do not preserve the order relationship among original data items. 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 quitely different depending on the input data. Thus, it is important to adopt proper encoding methods on account of data properties.

In the semi-adaptive encoding method, *huffman encoding* [15] and *arithmetic encoding* [24] are the examples.

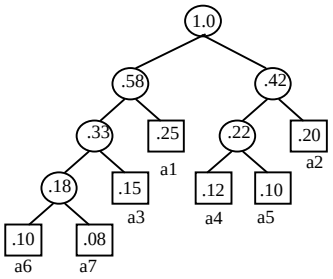


Figure 1: An example of the huffman tree

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 by 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 code generated by huffman encoding does not keep the order information among symbols.

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. These methods dynamically update statistics of each symbol based on the previous statistics (see details in [18]).

Recently, some research on compressors for XML data has 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*.

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.

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.

```
<book>
  <author> author1 </author>
  <title> title1 </title>
  <section>
    <title> title2 </title>
    <subsection>
      <subtitle> title3 </subtitle>
      ...
    </subsection>
  </section>
</book>
```

Figure 2: An example of XML data

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 *subsection* element is *book.section.subsection*.

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 Interval_T. The size of 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.

EXAMPLE 1. *Suppose that the frequencies of elements = {book, author, tile, 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:*

element	frequency	cumulative frequency	Interval_T
book	0.1	0.1	[0.0, 0.1)
author	0.1	0.2	[0.1, 0.2)
title	0.1	0.3	[0.2, 0.3)
section	0.3	0.6	[0.3, 0.6)
subsection	0.3	0.9	[0.6, 0.9)
subtitle	0.1	1.0	[0.9, 1.0)

Next, reverse arithmetic encoding encodes the simple path $P = p_1.\ldots.p_n$ of an element e into an interval $[min_e, max_e)$ using the algorithm in Figure 3.

Function reverse_arithmetic_encoding($P = p_1.\ldots.p_n$)
begin
1. $[min_e, max_e) := \text{Interval}_{p_n}$
2. **if**(n = 1) **return** $[min_e, max_e)$
3. length := max_e - min_e
4. $[q_{min}, q_{max}) :=$ reverse_arithmetic_encoding($p_1.\ldots.p_{n-1}$)
5. $min_e := min_e$ + length* q_{min}
6. $max_e := min_e$ + length* q_{max}
7. **return** $[min_e, max_e)$

end

Figure 3: An algorithm of reverse arithmetic encoding

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) in 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 $[q_{min}, q_{max})$ 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.

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:*

element	simple path	Interval_T	subinterval
book	*book*	[0.0, 0.1)	[0.0, 0.1)
section	*book.section*	[0.3, 0.6)	[0.3, 0.33)
subsection	*book.section.subsection*	[0.6, 0.9)	[0.69, 0.699)

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 search of Interval_Ts. 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 [21, 22] 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).

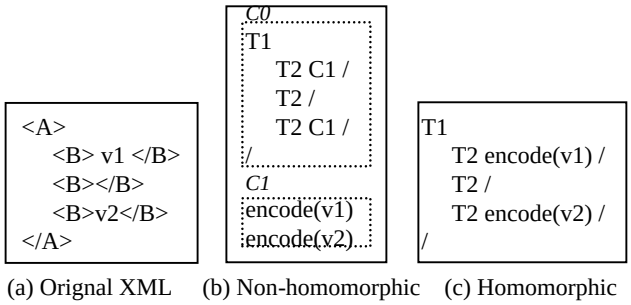
If the adaptive compression such as *adaptive huffman encoding* is applied, the compression time is saved since the preliminary scan is not required. In the adaptive compression, the encoded value of a certain symbol is changed depending on the location of the occurrence of the symbol since the adaptive 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 v1, v2 and their corresponding compressed version be c1, c2, then $v1 > v2 \not\Rightarrow c1 > c2$. 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.



(a) Orignal XML (b) Non-homomorphic (c) Homomorphic

Figure 4: An example of homomorphism

Like XGrind, XPRESS obeys the homomorphism [23]. 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. 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 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.

As a result, based on the above features, the compressed XML data generated by XPRESS supports the query processing 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.

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.

```
Function XML_Analyzer()
begin
1.  Pathstack := new Stack();
2.  Elemhash := new Hash();
3.  do {
4.      Token := XMLParser.get_Token()
5.      if(Token is a tag)
6.          Statistics_Collection(Token, Pathstack, Elemhash)
7.      else //Token is a data values
8.          Type_Inferencing(Token, Pathstack, Elemhash)
9.  }while(Token != EOF)
10. return Elemhash
end
```

Figure 6: An algorithm of XML Analyzer

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

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

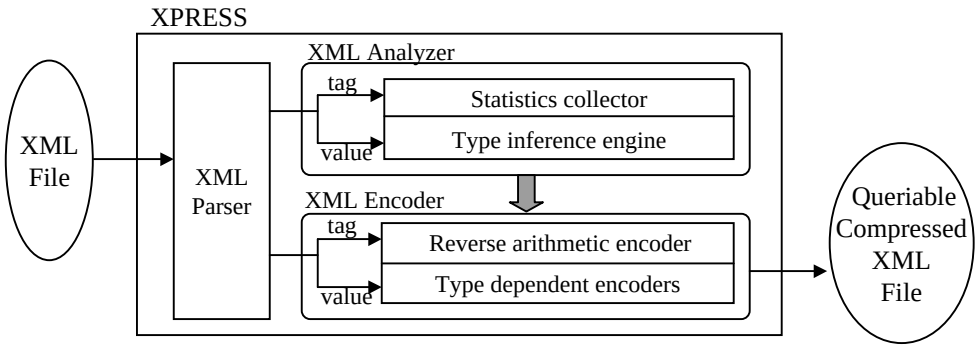
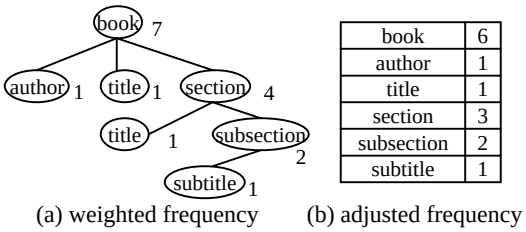


Figure 5: The architecture of XPRESS

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_T 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.



book	6
author	1
title	1
section	3
subsection	2
subtitle	1

Figure 7: Various frequencies

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 [1]. 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 [1], 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 [1], 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 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 *weighted frequency*, the intervals for higher level elements are enlarged. We call this frequency the *adjusted frequency*.

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 (taginfo = 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
```

Figure 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 five fields: *inferred_type, min, max, symhash* and *chars_frequecy*. 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. 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.

```
Procedure Type_Inferencing(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.man, intvalue)
12.          eleminfo.symhash.insert(Token)
13.          eleminfo.accumulate_chars_frq(Token)
14.       }
15.       else { // string
16.          eleminfo.symhash.insert(Token)
17.          if(the number of entries in eleminfo.symhash < 128) {
18.             eleminfo.inferred_type := enumeration
19.          }else eleminfo.inferred_type := string
20.          eleminfo.accumlate_chars_frq(Token)
21.       }
22.       break
23.    case enumeration :
24.       ...
25.       break
26.    case string :
27.       eleminfo.accumlate_chars_frq(Token)
28.       break
29. }
end
```

Figure 9: The algorithm of the type inference engine

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) 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 such that: 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. 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 string (Line (15)-(21)), we change inferred_type. Even though the preceding data values are integers, we change 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 string (Line (12) and (16)). Thus, if the number of distinct entries of symhash is less than 128, we assign *enumeration* to inferred_type. Otherwise, we assign *string* to inferred_type. Also, because inferred_type can be changed to *string*, chars_frequency field is updated (Line (20)).

When inferred_type is *enumeration* (Line (23)-(25)), we only check whether inferred_type can be changed to *string* without considering the type of the given data value. Thus, Line (24) is the same as Line (16)-(20). If inferred_type is *string*, chars_frequency field is updated only (Line (26)-(28)).

For brevity, we omit the behavior for the floating type. However, the extension of the algorithm for the floating type is straightforward.

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 1. Each distinct element has its own encoder which is one of six encoders.

Encoder	Description
u8	encoder for integers where max-min< 2^7
u16	encoder for integers where $2^7 + 1$ <max-min< 2^{15}
u32	encoder for integers where $2^{15} + 1$ <max-min< 2^{31}
f32	encoder for floating values
dict8	dictionary encoder for enumeration typed data
huff	huffman encoder of textual data

Table 1: 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.

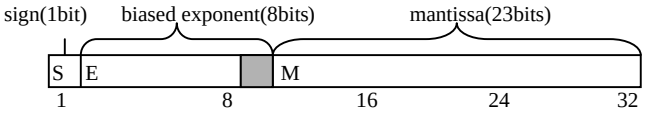


Figure 10: IEEE 32bit floating point standard 754

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.

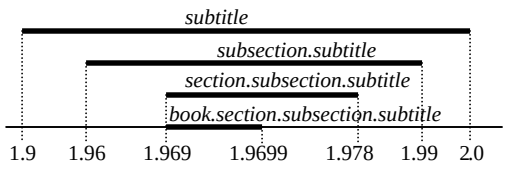
Start tags of individual elements are encoded by reverse arithmetic encoding using simple paths. In practice, we implement an approximated encoder, called the approximated reverse arithmetic encoder (ARAE), to improve the compression ratio and to parse compressed XML data without ambiguity.

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 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 $S \times 1.M \times 2^{E-127}$. 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 111**1**), 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 × 0.69 = 1.969 , 1.0 + 0.9 + 0.1 × 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.

```
Procedure XMLEncoder(Elemhash)
begin
1.  XMLParser.reinit()
2.  Initialization(Elemhash)
3.  Pathstack := new Stack()
4.  IntevalStack := new Stack()
5.  do {
6.     Token := XML_Parser.get_Token()
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
```

Figure 11: The algorithm of XML Encoder

The algorithm of XML Encoder is in Figure 11. First, XML Parser 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 for end tags and the value less than 2^{-15} can not be represented using 15 bits. Also, for the same reason, we adjust the length of $Inteval_T$ 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 partitions a long label path expression into short label path expressions whose corresponding interval sizes are greater than 2^{-15} using ARAE. Thus, a label path expression is transformed into a sequence of intervals. Generally, the length of the sequence is 1 since a label path expression is usually short. By using the sequence of intervals, the query executor tests elements in compressed XML data whether their encoded values are in an interval of the sequence or not.

Data values of exact matching conditions in a query are converted into encoded values. Then, the query processor detects the elements which satisfy the label path expression and the value without decompression. Also, the range

[2]it is represented by the gray box in Figure 10

condition for a numeric typed element are encoded by the data value encoder for the element. Then, without the decompression of encoded values, the query is evaluated. For the range condition for 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.

5. EXPERIMENTS

To show the effectiveness of XPRESS, we empirically compared the performance of XPRESS with two representative XML compressors XMill[3] and XGrind[4] 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. XPRESS shows significantly better query performance than XGrind.

5.1 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 Shakespeare. The characteristics of the data sets used in our experiment are summarized in Table 2. 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 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.

Data Set	Size	Depth	Tags	Numeric	Enum
Baseball	17.06	6	46	19	5
Course	12.28	6	18	5	4
Shakespeare	15.3	5	21	0	0

Table 2: XML Data Set

The Baseball [13] 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. To test the effectiveness of large sized XML data, we scale up the original data by 16 times.

The Course [2] 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. We scale up the original data by 4 times.

The Shakespeare [7] is the collection of plays of Shakespeare which is marked up by Jon Bosak. In our experiment, we concatenated 37 plays of Shakespeare into a single XML document. And, we scale up the original data by 2 times.

Queries We evaluated XPRESS using several queries. The characteristics of queries used in our experiment are described in Table 3.

The first character in the first column indicates the data set on which the query is executed: 'S' denotes the Shakespeare, 'B' is for the Baseball and 'C' is for the Course. The number in the Query name 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 partial matching path queries with the range of data values. Query definition in Table 3 describes the corresponding XPath 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 enumeration typed data values, C4 is the range query of the integer typed data values and S4 is the range query of the general textual data values.

5.2 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. Lastly, we show the query performance of XPRESS with that of XGrind.

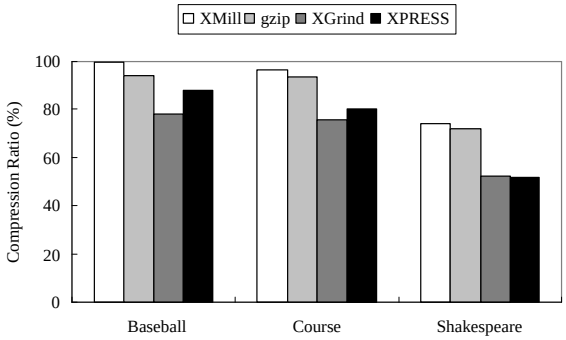


Figure 12: Compression ratio

Figure 12 shows the compression ratios for different data sets and compressors. For each data set, the four connected bars represent XMill, gzip, XGrind and XPRESS. Since XMill uses the dictionary encoding method for struc-

[3]available in http://www.researech.att.com/sw/tools/xmill/
[4]available in http://sourceforge.net/projects/xgrind/

Query name	Query definition
B1	/SEASON/LEAGUE/DIVISION/TEAM/PLAYER/GIVEN_NAME
B2	//TEAM/PLAYER/SURNAME
B3	/SEASON/LEAGUE//TEAM/TEAM_CITY
B4	/SEASON/LEAGUE//TEAM[TEAM_CITY >= Chicago and TEAM_CITY <= Toronto]
C1	/root/course/selection/session/place/building
C2	//session/time
C3	/root/course//session/time/start_time
C4	/root/course//session/time[start_time >= 800 and start_time <= 1200]
S1	/PLAY/ACT/SCENE/SPEECH/STAGEDIR
S2	//PGROUP/PERSONA
S3	/PLAY/ACT//SPEECH/SPEAKER
S4	/PLAY/ACT//SPEECH[SPEAKER>= CLEOPATRA and SPEAKER <= PHILO]

Table 3: XML Query Set

tural information, and groups semantically related data values into containers before compressing with zlib, as we expected, XMill achieved the best compression ratio, on the average of 92%. The average compression ratio of XPRESS is 73%. 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 2, the Shakespeare does not contain any numeric and enumeration typed data. For the Shakespeare data set, the compression ratio of XPRESS is slightly lower than that of XGrind since the huffman encoder in XPRESS inserts the length of an encoded value in front of the value. However, XPRESS shows a reasonable compression ratio for all cases.

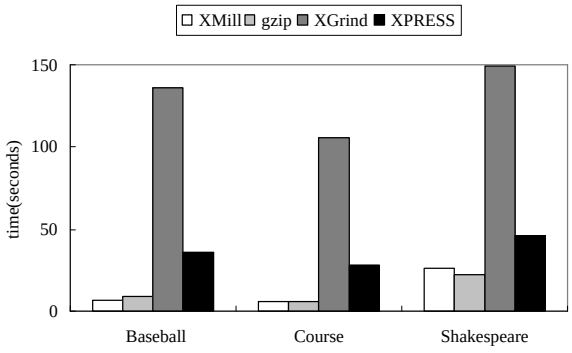


Figure 13: Compression time

Figure 13 shows the compression time of each compressor. 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 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 dose 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 of the compression time. Thus, we omit the graph of the decompression time.

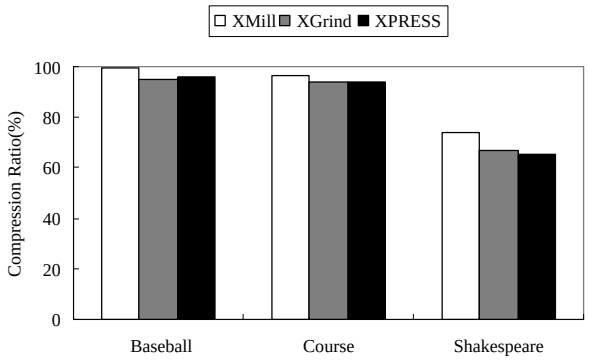


Figure 14: Compression ratio after performing gzip

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 14. In Figure 14, 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 in Figure 15. The query performance of XPRESS outperforms that of XGrind over all cases.

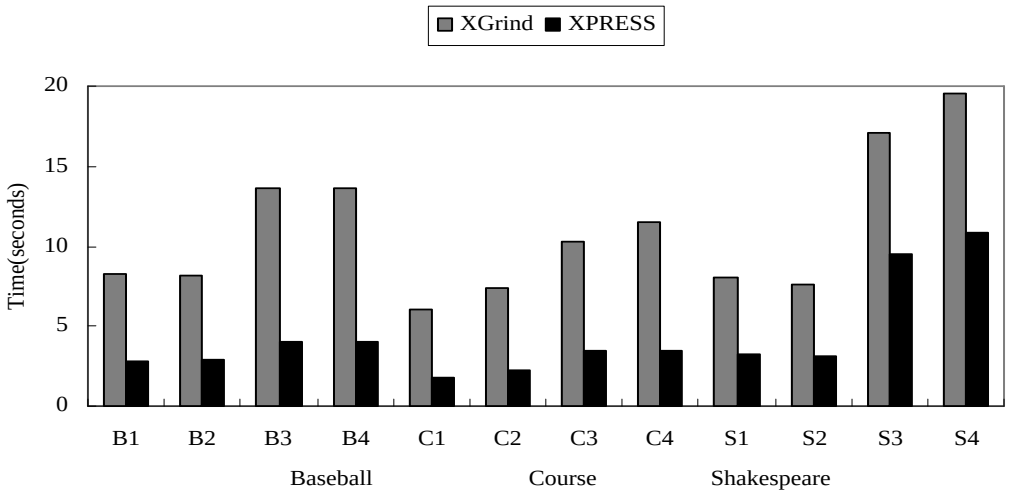


Figure 15: Query evaluation time

The query cost of query type 1 (i.e., B1, C1, and S1) 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 S2) are short and simple, the query processing cost is cheaper than those of query type 3 and query type 4. Thus, the difference of query performance of query type 2 between XPRESS and XGrind is less conspicuous than those of query type 3 and query type 4. However, the query performance of XPRESS for complicated path expressions (B3, C3, and S3) 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 S4. 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 2.83 times better than that of XGrind.

Consequently, XPRESS achieves significantly improved query performance compared to XGrind and shows the reasonable compression ratio.

6. CONCLUSION

In this paper, we propose XPRESS, an XML compressor which supports direct 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 minimize the overhead of a partial decompression for range queries.

We implemented XPRESS and a query 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 2.83 times better than that of an existing XML compressor and the compression ratio of XPRESS is 73%.

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 Resouce Identifier) using data mining algorithms.

7. ACKNOWLEDGMENTS

This work was supported in part by Brain Korea 21 Project and in part by the ministry of information and communication, Korea, under the Information Technology Research Center(ITRC) Support Program. The support of Sun-Hwa Lee is greatly appreciated.

8. REFERENCES

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 591–600, September 2001.

[2] Anonymous. http://www.cs.washington.edu/research/projects/xmltk/www/xmlproperties.html.

[3] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. Working Draft, http://www.w3.org/TR/2002/WD-xquery-20020816, 16 August 2002.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0. W3C Recommendation, http://www.w3.org/TR/REC-xml, 1998.

[5] C.-W. Chung, J.-K. Min, and K. Shim. APEX: An Adaptive Path Index for XML Data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 121–132, June 2002.

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

[7] R. Cover. The XML Cover Pages. http://www.oasis-open.org/cover/xml.html, 2001.

[8] M. F. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of the 14th International Conference on Data Engineering*, pages 14–23, February 1998.

[9] M. F. Fernandez, W. C. Tan, and D. Suciu. SilkRoute: trading between relations and XML. *WWW9/Computer Networks*, 33(1-6):723–745, June 2000.

[10] D. Florescu and D. Kossman. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, September 1999.

[11] R. Goldman and J. Widom. DataGuides: Enable Query Formulation and Optimization in Semistructured DataBases. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445, August 1997.

[12] T. Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 109–120, June 2002.

[13] E. R. Harold. Long Baseball Examples from The XML Bible. ibiblio, http://www.ibiblio.org/xml/examples/baseball/.

[14] P. G. Howard and J. S. Vitter. Analysis of Arithmetic Coding for Data Compression. In *Proceedings of the IEEE Data Compression Conference*, pages 3–12, April 1991.

[15] D. A. Huffman. A Method for the Construction of Minimum Redandancy Codes. In *Proceedings of the Institute of Radio Engineers 40*, pages 1098–1101, September 1952.

[16] H. Liefke and D. Suciu. XMill: An Efficient Compressor for XML Data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, May 2000.

[17] C.-W. Park, J.-K. Min, and C.-W. Chung. Structural Function Inlining Technique for Structurally Recursive XML Queries. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 83–94, August 2002.

[18] D. Salomon. Data Compression, the complete reference. Springer-Verlag New York, Inc, 1998.

[19] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of 26th International Conference on Very Large Data Bases*, pages 65–76, September 2000.

[20] C. E. Shannon. A Mathematical Theory of Communication. *Bell Syst. Tech. J.*, 27:398–403, July 1948.

[21] T. Shimura, M. Yoshikawa, and S. Uemura. Storing and Retrieval of XML Documents using Object-Relational Databases. In *Proceedings of 10th International Conference, DEXA*, pages 206–217, August 1999.

[22] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215, June 2002.

[23] P. M. Tolani and J. R. Haritsa. XGRIND: A Query-friendly XML Compressor. In *Proceedings of 18th International Conference on Database Engineering*, February 2002.

[24] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, June 1987.