IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 17, 2021, accepted April 5, 2021, date of publication April 13, 2021, date of current version April 20, 2021.

*Digital Object Identifier 10.1109/ACCESS.2021.3073041*

# Compression of XML and JSON API Responses

**GYAN P. TIWARY[1], ELENI STROULIA[2], (Member, IEEE), AND ABHISHEK SRIVASTAVA[1]**
[1]Discipline of Computer Science and Engineering, IIT Indore, Indore 453552, India
[2]Faculty of Computing Science, University of Alberta, Edmonton, AB T6G 2R3, Canada

Corresponding author: Abhishek Srivastava (asrivastava@iiti.ac.in)

This work was supported in part by the Ministry of Electronics and Information Technology (MeiTY), Government of India, in part by the Science and Engineering Research Board (SERB), Government of India, and in part by the University of Alberta, Edmonton, Canada.

**ABSTRACT** Web services are the de-facto standard for implementing web-based systems today, and comprise message-based interactions involving XML and JSON documents. These formats can be quite verbose, especially XML, and therefore compression of such documents can potentially improve the communication efficiency and performance of service-oriented systems. In this paper, we review the various formulations of XML compression and propose a novel technique for the same, wherein large section of the documents are substituted by numerical representations. The approach is simple yet effective, especially on small documents that constitute the bulk of communicated content in web-based systems. We conduct experiments with several datasets and demonstrate that the proposed technique for XML compression outperforms the existing state-of-the-art techniques.

**INDEX TERMS** Compression of Web API responses, compression of ReSTful responses, JSON compression, XML compression.

## I. INTRODUCTION

In service-oriented architectures, data exchange relies on some choice of resource representation, with XML or JSON being the most popular ones. XML and JSON are structured documents where structure tags specify the type of the data elements they annotate, enabling the independent services to share commonly agreed upon semantics. Element and attribute tags in XML and, correspondingly, keys in JSON serve as annotations to the actual data values being exchanged. These annotations tags are invariably repeated several times in the request and, especially response, documents. While this repetition arguably makes XML and JSON self-descriptive, it also contributes to verbosity of these documents. The main purpose of compressing structured documents such as XML and JSON is to reduce their size without information loss.

Consider for example an API returning information about a flight to `Edmonton`, at `16:02:30` on `2019-12-24` at gate `6`, having departed at `16:22:30` from gate `7`, shown in listing 1 as XML and in listing 2 as JSON. The size of the XML representation is 169 bytes and the JSON representation is slightly smaller at 154 bytes, whereas the size of the data itself is only 55 bytes. In general, the sizes of the XML and corresponding JSON representation of the same structured data returned by a web service are comparable, with JSON being slightly smaller because it does not require ''closing tags'' for the various data elements.

In our literature review on compression of structured documents, most work has been on the compression of XML and relatively limited attention has been given to JSON. The method we describe in this paper is applicable to structured documents in general, including XML and JSON, relying on the notion that the documents in question exhibit a regular structure composed of free text encapsulated in regular patterns of labels, i.e., words from a limited vocabulary. In this paper, we report on our experiments with XML, primarily because this enables us to compare our method against other related methods that have been validated with XML. To the best of our knowledge, there has been no reports of JSON compression in the literature. A detailed discussion of the methods of compression is given in section II. Since most of the work is done on XML compression and experimental datasets are available only in XML compression, our evaluation is based on XML compression only.

An XML document (as well as a JSON document) is structured as a tree, with each node labelled by a tag consisting of more children nodes and/or plain data, conveying the actual information content. In the example shown in Listing 1 ''Edmonton'', ''16:02:30'', ''16:22:30'', ''2019-12-24'',

The associate editor coordinating the review of this manuscript and approving it for publication was Zhangbing Zhou.

```
<Airport City="Edmonton">
  <Arrival-time>16:02:30</Arrival-time>
  <Departure-time>16:22:30</Departure-time>
  <Date>2019-12-24</Date>
  <Gate>6</Gate>
  <Gate>7</Gate>
</Airport>
```

**Listing 1.** API response in XML.

```
{ "Airport":
  { "Arrival-time": "16:02:30",
    "Departure-time": "16:22:30",
    "Date": "2019-12-24",
    "Gate": "6",
    "Gate": "7",
    "_City": "Edmonton", }
}
```

**Listing 2.** API response as a JSON.

''6'' and ''7'' constitute the plain data, i.e., the actual information to be communicated. The element and attributes tags, i.e., ''Airport'', ''City'', ''Arrival-time'', ''Departure-time'', ''Date'' and ''Gate'', constitute the structure of the document.

XML documents are text files and, in principle, they can be compressed using simple text compressors [2], [3]. Such text compressors, however, are XML-agnostic and ignore the structured nature of XML. Therefore, even though ordinary text compressors do compress XML documents, taking the structure of the document into consideration can result in significant reduction in the size of the compressed documents.

The main objective of this research is to minimize the size of the XML or JSON documents being communicated between web services. Web services provide a programmatic way of accessing the web. There are two ways of implementing web services, SOAP based web services and RESTful web services. SOAP Based Web Services define the XML schema for their request and response documents. Restful Web Services do not require the precise definition of the structure of their requests and responses. Generally, the size of the XML (or JSON) exchanged in either type of web services is not very large and these have a high structure-part/data-part ratio. This paper proposed a technique to more effectively compress such small XML or JSON documents used in web APIs.

Compression techniques that take advantage of the structured nature of XML are called XML-conscious compressors [1]. XMill [4] is one among the earliest XML-conscious compression algorithms and is often used as the benchmark against which other methods are evaluated. XML-conscious compressors are further divided into two categories, based on whether or not they produce queryable compressed documents. Queryable XML compressors (partially) maintain the structure of the original document in the compressed result, such that XQuery, and other similar types of querying mechanisms, may extract data without decompressing the whole document. These methods are particularly relevant in XML-storage systems to support space-efficient storage and time-efficient access. XMill [4] is a non-queryable compressor and XGrind [5] is the oldest and perhaps best known XML compressor of the queryable type.

Another approach to XML compression involves aggregation and clustering of XML documents wherein several small XML documents are aggregated to form a larger one, which is subsequently compressed in XML-conscious manner. SMCA [6] is a state-of-the-art techniques in this direction. While XMill is the most prominent technology of XML compression, SMCA is one of the latest research in the field of SOAP based XML message compression. The classification of XML compressors and related research is described in detail in the Section II.

To understand the problem statements of the proposed research, it is necessary to understand the workings of XMill [4] and SMCA [6], the most significant XML compression techniques. SMCA is largely based on XMill and the points below summarize the common features shared by both of them.

- Both techniques treat the data and structure parts of XML differently.
- In both techniques, a 'tag dictionary' assigns numbers to the structure parts of the XML based on their position in the document. For example, the tag dictionary corresponding to the XML in Listing 1 comprises six tag names, i.e., Airport, City, Arrival-time, Departure-time, Date and Gate, and their respective codes, i.e., 1, 2, 3, 4, 5 and 6.
- In both techniques, a 'path' to each XML data (content) point is worked out. The path to a data point in XML is the sequence of tag names from the root tag to the parent tag of the data element in question. For example, the data point ''2019-12-24'' in Listing 1 has the following path: ''Airport/Arrival-time''. Data points ''6'' and ''7'' have the same path i.e., ''Airport/Gate''. An important assumption of both XMill and SMCA is that data points with the same data path are similar.
- Both XMill and SMCA create a new container for every new path and insert all the data that share a common path a common corresponding container.
- The major premise of both XMill and SMCA is that compressing similar data together in a container with compressors like bgip2 [2], provides better compression.

Assuming that compressing an individual XML documents does not yield a sufficient compression ratio, SMCA works on several similar XML documents, aggregated together. In SMCA, the contents from multiple XML elements with the same path are put together in one container to get better compression. Both XMill and SMCA suffer from some the following drawbacks.

- Certain types of XML documents result in the creation of many containers, each with a small number of data elements, a phenomenon which adversely affects the overall compression ratio. The XML documents used

in Web APIs (either SOAP or RESTful) tend to suffer from this problem, since they are typically short and their data elements do not share common paths. XMill and SMCA are, therefore, less effective in compression of short XML documents.
- Assigning numbers to the structured parts of the XML document based on their order in which they appear in the document may result in infrequent structured elements being assigned smaller numbers, i.e., numbers with fewer digits, than more frequent elements. This results in bulkier than necessary path containers, and therefore reduces the effectiveness of the compression process.
- Neither XMill nor SMCA maintain the XML structure in the compressed format, and, therefore, the compressed document is not queryable.

In view of the drawbacks of existing XML compression techniques, we raise the following hypotheses.

- A small number of containers with a large number of similar data elements can result in better compression.
- Better compression can be achieved if the structure tags of the XML document are assigned the smallest possible number (starting from 1) in the tag dictionary based on their probable frequency. More frequent tags should be given small numbers, and vice-versa.

Using these hypothesis, in this paper, we describe a new XML compression technique motivated primarily by the need to enable bandwidth-efficient communications within service-oriented systems and aims to reduce the size of the XML documents exchanged as requests and responses between web APIs. The size of these documents is typically not very large (generally less than 1MB) and their content typically involves a high ratio of element and attribute tags to data content.

We have evaluated our method using the dataset used by the authors of the SMCA compression method [6] that contains a set of small XML API-response documents, and larger XML documents constructed by aggregating the former ones. Our compression method outperforms the state-of-the-art techniques in terms of compression ratio for short messages ($<$ 1MB). Its performance is comparable to the best in literature for large messages ($>$ 1MB) with the added advantage that our technique results in queryable compressed documents. We also evaluated our technique against XMill, on the dataset used in [1]. This dataset comprises of very large XML files, containing mostly textual data, annotated with XML tags. These documents are typically the result of manual or automated text processing workflows that embed analysis results in the text, as structured tags. In contrast to database-driven API responses, these documents tend to have lower element and attribute tag ratio relative to data. Again, our method outperforms XMill in most cases on this dataset.

The rest of the paper is organized as follows. Section II discusses the classification of XML compression techniques and provides an overview of previous research on XML compression. The technique we propose is discussed in detail in Section III. In Section IV, we evaluate the proposed technique and compare it against other techniques in the literature. The proposed technique of XML compression is not insulated from limitations, which we discuss in Section V. Finally we conclude the paper and discuss possible avenues for future research in Section VI.

## II. RELATED WORK

In principle, compression techniques belong in one of two categories: *general-text compressors* and *XML-conscious compressors*. XML is a text file, and like any other text file, it can be compressed by any general text compressor such as bzip2 [2], ppm [3]. A general text compressor, however, does not take advantage of the special features of XML and, therefore, tends to result in inferior compression ratio. XML-conscious compressors, on the other hand, utilise the structure of XML documents and therefore are able to compress XML documents with better compression ratios. XMill, XGrind and SMCA [6] are examples of XML-conscious compressors.

XML-conscious compressors are next divided into three categories: *schema-dependent*, *schema-independent* and *XML tree-based* compressors. Schema-dependent compressors require the schema of the XML documents to effectively compress a document. rngzip and [8] are examples of schema-dependent compressors. Schema-independent compressors, on the other hand, do not require the schema. XMill, XGrind and SMCA are examples of schema-independent compressors. Finally, XML tree-based compressors take advantage of the fact that XML documents are organized in a tree-like structure and aim to make this tree smaller. This is done by representing the XML trees as a DAG (Directed Acyclic Graph) or a FCNS (First Child Next Sibling) list: [9], [10] and [11] are examples of XML tree-based compressors.

Furthermore, based on whether or not the resulting compressed documents support queries such as XQuery, compressors may be categorized as *queryable XML compressors* and *Non-Queryable XML Compressors*. XGrind is a queryable compressor while XMill and SMCA are non-queryable compressors. Queryable compressors can be *homomorphic* and *non-homomorphic*. The former type retain the original structure of the XML even after compression, whereas non-homomorphic compressors do not. XGrind is a homomorphic compressor while XMill and SMCA are non-homomorphic compressors. The XML compressor proposed in this paper is XML-conscious, schema independent, queryable, and homomorphic.

Much of the research done so far on XML compressors is around XMill. XMill separates the data and structure parts of the XML and inserts the data values into different containers based on the path and data type. A dictionary is created to assign a unique number to every unique element and attribute name. Each container is then compressed using a back-end compressor specialized for that data type. Most general-purpose compressors can be used as back-end

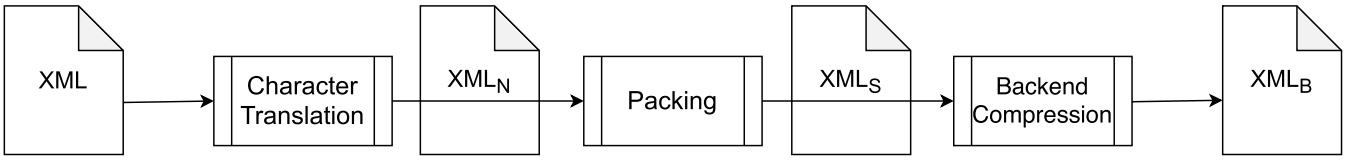


**FIGURE 1.** The three steps of the compression process.

compressors. XGrind is a two-pass compressor: the output of XGrind is structured along the lines of its input XML. Querying or parsing thus becomes possible even on the compressed documents. XGrind encodes the element names and attributes by numbers followed by 'T' and 'A' ('T' for tag, 'A' for attribute) and compresses the values using Huffman coding. XML does not lose its basic structure due to compression with Huffman coding, and this feature makes XGrind queryable.

XGrind, like XMill, is a seminal piece of work on XML compression and several research endeavours have built on it. Prominent among these are [12], [13] and [14]. Al-Shammary and Khalil [15] was the first attempt at XML compression keeping in mind SOAP-based web services. In this work, the XML tree is first converted into a binary tree, using a FCNS (First Child Next Sibling) transformation. Subsequently, a sibling state of two bits to each node of the binary tree is attached. Finally a Huffman code is assigned to each node and the entire XML is compressed. The resulting compressed document is non-queryable.

This was followed by an era of aggregation-based research on SOAP messages. This line of research is based on the principle that, if a group of XML documents is compressed instead of compressing each XML message separately, the compression ratio will be better [16]–[23]. SMCA [6] is perhaps the most recent major research endeavour in grouping based XML compression, outperforming most approaches of XML compression in terms of compression ratio. It is the only technique that aggregates and compresses multiple XMLs in one pass and generates a single non-queryable output. However, RESTful services are known for single request and response, so aggregating multiple XMLs is not relevant in the case of RESTful services.

Various XML compressors discussed in this section are not as effective at compressing relatively small (<1MB) XML documents in terms of compression ratio. XML documents of this size are mostly utilised in API requests and responses. The norm in ReSTful web services is that a single request is followed by a single response. Therefore the technique of grouping responses to several requests together is somewhat impractical. The method we propose in this paper overcomes the shortcomings of earlier approaches and is especially effective for relatively small XML documents and is therefore ideal for use with API request and responses (especially ReSTful Web services). Further, the proposed technique creates queryable compressed documents.

## III. THE PROPOSED TECHNIQUE

In this section, we describe in detail our method for XML compression. The proposed method deals with all the problem statements discussed in Section I. It is designed to examine the hypotheses discussed in Section I. Our method converts the entire XML into a special type of binary encoding in a single container. A back-end compressor is then applied to the entire container. The structure parts of XML are assigned with a smallest possible number (starting from 1) in the tag dictionary according to their probable frequency in XML. The tag dictionary is referred to as "tag table" in this article. The tag table for the XML in Listing 1 is shown in the Table 2.

Note that our method relies on a simple key idea: each element and attribute tag is mapped to a number; all closing tags are also mapped to another single number; finally, each distinct symbol of the language, in which the free-text of the document is written, corresponds to its own number. This translation process applies equally to XML and JSON documents, since the only difference between the two notations is in the lack of pairs of opening/closing tags in JSON.

The method involves a *configuration* and a *compression* phase. The *configuration* phase creates two dictionaries, the *symbol table* (i.e. Table 1) and the *tag table* (i.e. Table 2), which are used for the *compression* process, shown in Figure 1.

The first step of the *compression* process is the *character translation*, wherein the original XML document is converted into $XML_N$, a form comprising only 13 symbols that can be represented by just 4 bits. In the subsequent *packing* step,

**TABLE 1.** A subset of the symbol Table.

| . | .. |
|---|---|
| E | 38 |
| d | 69 |
| m | 78 |
| o | 80 |
| . | .. |

**TABLE 2.** The Tag Table corresponding to the flight-description XML document.

| | |
|---|---|
| Airport | 2 |
| Arrival-time | 3 |
| City | 4 |
| Date | 5 |
| Departure-time | 6 |
| Gate | 1 |

every two contiguous symbols of $XML_N$ are accommodated in one byte. This is done for all symbols contained in $XML_N$ and the resulting document is called $XML_S$. Finally, the third *back-end compression* step involves compression of $XML_S$ using any one of the various back-end compressors like PPM [3] to form the $XML_B$ document.

### A. CONFIGURATION

The *configuration* phase creates the *symbol table*, which maps each character present in the data part of the input XML document to a unique number, and the *tag table*, which maps each element and attribute tag-name into a unique number.

According to a recent survey [7], 60.7% content on the web are in English. The second most used language is Russian, which is 8.3% of total and third most used language on the web is Turkish which is just 3.9% of total. In addition, several other languages use the Latin script. We are not entirely sure about the programmable web, but realistically we expect the language distribution of APIs to be along similar lines. Therefore, assuming English as the main language of XML documents, the document characters are in the range of 32 to 127 in the ASCII table. The symbol table, therefore, is a mapping of each character in this range of ASCII values to a unique number. Each character in the symbol table is mapped to a two-digit number, computed as the *(ASCII value of the character - 31)*. Table 1 shows a subset of the symbol table. The symbol table is independent of the specific XML document under compression; instead it only depends on the language of the document. Therefore, the same symbol table is used to compress all XML documents in a given language. Given the total number of characters in the English alphabet, commonly used symbols, and the Arabic numerals, the range of the Symbol Table values would be covered by two-digit numbers.

Tag tables depend on the XML schema underlying the to-be-compressed document, which, in turn depends on the web API. In principle, a tag table could be constructed once and reused for all documents emitted by the API. However, to accommodate XML documents that are the result of text annotation and to take advantage of the possibility that a particular response may not use all the tags in the schema, the tag table is constructed by scanning the XML document and mapping every unique element and attribute to a unique number. More frequent tags are mapped first to smaller numbers, whereas tags that appear less frequently are mapped to subsequent larger numbers.

To illustrate the construction of the tag table, we use the XML shown in listing 1. Table 2 shows the corresponding tag table. As ''Gate'' is the most frequently appearing tag in the XML, it is mapped to the smallest number; all tags and attributes are also mapped to unique numbers.

Considering the practicalities of the deployment of our compression method, we propose that service providers share the symbol table and tag table of their XML response documents, along with the documentation of their APIs.

```
T2 A4 1314150616070616 T3 2526272829273028 0
T6 2526272929273028 0 T5
    29282534232529232935
0 T1 23 0 T1 24 0 0
```

**Listing 3.** Example of the $XML_N$.

### B. COMPRESSION

The actual compression process is carried out in three steps: *character translation*, *packing*, and *back-end compression* as shown in Figure 1.

#### 1) CHARACTER TRANSLATION

In this step, the symbol table is used to translate the data part of the XML document, one character at a time. The structure part of the XML is translated using the tag table, with an entire element or attribute tag substituted by their corresponding number in the tag table. The intermediate file produced by this step is called $XML_N$. The $XML_N$ corresponding to the XML document in the Listing 1 is shown in the Listing 3. In the $XML_N$ 'T' denotes the tag name, the number 2 following T is the tag table entry corresponding to the word ''Airport'' in the XML. Similarly, 'A' represents an attribute name and the number following A, 4 in this case, is the tag table entry corresponding to the word ''city'' in the XML. Zero in $X_N$ indicates the closing tag for the last (or most recently opened) tag. The remaining numbers that are neither zero nor have an 'A' or 'T' before them represent the data part of the XML. Each character of the data parts of the XML is substituted with the help of the symbol table. In this way, $XML_N$ is made up of only 13 types of symbols, the Element Start Symbol (T), the Attribute Symbol (A), Space, and the digits 0 to 9. Element End symbol is denoted by a single 0 preceded and followed by Space. Therefore no extra symbol is required to denote the end of an element.

The character-translation process is described formally in Algorithm 1. Every token of XML is picked up one by one in different iterations (Line 3). If the chosen token in the current iteration is a data part of the XML (Line 4), each character of that token is translated with the help of the symbol table, and stored in the variable *translate* (Line 5). If the chosen token is a structure part of the XML (Line 6), the entire token is translated to a minimum possible number with the help of the tag table, and stored in the variable *translate* (Line 7). If the chosen token in the current iteration is a tag of the XML (Line 8), the character 'T' is prepended to the current value of *translate* (Line 9). If the chosen token in the current iteration is an attribute-name of the XML (Line 10), the character 'A' is prepended to the current value of *translate* (Line 11). If the chosen token in the current iteration is a closing tag of the XML (Line 12), the variable *translate* is set to 0 (Line 13). After obtaining a proper token it is concatenated with the variable $XML_N$ followed by a space (Line 14). As many iterations are performed as is necessary to translate all the tokens in the XML document, and return the value of $XML_N$.

**Algorithm 1:** Character_Translation_Algorithm

**Input:** Plain_XML
**Result:** $XML_N$

```
1  var XML_N;
2  var translate;
3  for All the tokens in Plain_XML do
4      if The token is a "data" part of the XML then
5          translate ← Use Symbol Table to translate each
             character of the token;
6      if The token is a "structure" part of the XML then
7          translate ← Use Tag Table to translate whole
             token;
8          if The word is a "tag" in the XML then
9              translate ← 'T' concatenate(+) translate;
10         if The word is a "attribute-name" in the XML
             then
11             translate ← 'A' concatenate(+) translate;
12     if The word is a "Tag Closing" then
13         translate ← '0';
14     XML_N ← XML_N + SPACE + translate;
15 end
16 return XML_N;
```

**Algorithm 2:** Packing_Algorithm

**Input:** $XML_N$
**Result:** $XML_S$

```
1  var XML_S ← null;
2  Assign all the 13 symbols in XML_N to an unique 4-bit
     binary;
3  var cursor ← First Symbol in XML_N;
4  for cursor is not end of XML_N do
5      if A character at cursor + 1 exists then
6          var cursor1 = cursor + 1 (i.e. next symbol);
7          Accommodate 4-bit binaries equivalent of
             symbols pointed by cursor and cursor1 into a
             single byte;
8          Concatenate the created byte with current value
             of XML_S;
9      if If cursor + 1 is end of the XML_N then
10         Create a byte using the 4-bit binary of character
             at cursor and 0000;
11         Concatenate the created byte with current value
             of XML_S;
12     cursor = cursor + 2;
13 end
14 return XML_S;
```

#### 2) PACKING

As the total number of symbols in $XML_N$ is just 13, the symbols may be represented using 4 bits. In the packing step, every two contiguous symbols of $XML_N$ are accommodated in one byte. Doing this for the entire content of $XML_N$ results a 'squeezed' binary file called $XML_S$. As both $XML_N$ and $XML_S$ retain the tree structure of the original XML, they may be parsed with the help of the symbol and tag tables and Xquery queries (or similar) can be executed on them.

Packing is described formally in Algorithm 2. Everything starts with mapping all the 13 different symbols used in $XML_N$ to a unique 4-bit binary (Line 2). A cursor, denoted by the variable *cursor*, points to the first symbol present in $XML_N$ (Line 3). If this cursor is not the end of the $XML_N$ (Line 4) and there exists another symbol next to where the cursor is pointing (Line 5), then the symbol next to the current cursor is pointed to by another cursor denoted by the variable *cursor1* (Line 6). The 4-bit binary mappings of the symbols being pointed by *cursor* and *cursor1* are accommodated in one byte (Line 7). However, if the variable *cursor* is pointing to the last symbol of the $XML_N$ (Line 9), then the byte will be created by assuming the next 4-bit binary as *0000* (Line 10). The created byte is appended to the variable $XML_S$ (Line 8 and 11). Because in an iteration we are picking two values from the $XML_N$, so for the next iteration the cursor has to be advanced by two symbols. Finally the algorithm returns $XML_S$.

#### 3) BACK-END COMPRESSION

The this final step of the compression process, the $XML_S$ created during the packing stage is passed to any one of various back-end compressors. In this work, we choose PPM [3] as the back-end compressor because of its superior compression ratio in our case. Any other general purpose compressor may also be used for back-end compression. The output of this step is called $XML_B$. Parsing $XML_B$ or executing queries on it is not possible. SMCA, XMill, and many other XML-compression techniques use back-end compressors. We too have used a back-end compressor to show the effectiveness of the proposed technique against existing techniques.

The XML schema is required to create the tag table. However, once this table is created, the entire XML document need not be handled together for effective compression. Different parts of an XML can be compressed separately using the tag and symbol tables. Just one tag name of a large XML document can be substituted using the symbol table and tag table to get the corresponding $XML_N$. This $XML_N$ that represents a very small portion of the document can be converted into $XML_S$ followed by $XML_B$.

### C. JSON COMPRESSION

The JSON document shown in Listing 2 conveys the same information and exhibits a parallel structure to the XML document in Listing 1, both consisting of a "Data" and a "Structure" part. The two representations can be easily converted to each other. Therefore, the character translation technique discussed in the section III-B can be applied to both representations and the output would be the same. However, most related research on compression is in terms of XML, and only XML datasets are available for comparative experimentation, therefore we have chosen to explain our work in

the terms of XML, fully cognizant of the fact that mostly web APIs exchange information in JSON.

### D. COMPRESSION OF NON-ENGLISH BASED XML

The effectiveness of the proposed XML compression technique depends to a large extent on the number of digits in the entries of the symbol table. This number depends on the number of unique characters that the XML is composed of. Each character in the symbol table is assigned a unique number. The number assigned to these characters solely depends on number of unique characters used in XML and is not related to the encoding of characters (e.g., UTF-8, UTF-16, or UTF-32). Similarly, the tag table contains mappings of element and attribute names of the XML to the smallest possible number based on the frequency of occurrence the concerned tag and is also not related to the encoding of characters in the tag. In the proposed method, we have only considered the English alphabet because of two reasons.

- Our work is motivated by the need to improve the performance of web applications through the compression of API responses. Given that 60.7% of the content on the Internet is in English [7] it stands to reason that most of the API responses use English characters. With this, one can safely assume that most XML documents comprise content only in English and this significantly helps compression as it makes the entries in the symbol table be of just two digits.
- If we consider XML documents with non-English characters, the number of digits required for the symbol table entries increase and the compression is not as effective. Finally, because earlier research on XML compression assumes English based content only, we too show compression results on XML documents in English.

The proposed method can be easily updated to work on non-English characters. If the maximum number of unique characters that an XML can have is increased to 999, which is enough to accommodate most non-English characters, then each character in the symbol table would be mapped to a number of three digits. The details on which three-digit number is to be mapped with which character in the symbol table, can be worked out at the time of implementation. One possible approach to render some efficiency could be to assign a small number to a character with a small Unicode and a large number to a character with a larger Unicode. A subset of such a symbol table is shown in the Table 3:

**TABLE 3. A subset of the Symbol Table having character to 3-digit number mapping.**

| | |
|---|---|
| . | .. |
| E | 038 |
| d | 069 |
| m | 078 |
| o | 080 |
| . | .. |

```
T2 A4 013014015006016007006016 T3
   025026027028029027030028 0 T6
   025026027029029027030028 0 T5
   029028025034023025029023029035 0 T1 023 0
    T1 024 0 0
```

**Listing 4. Example of the $XML_N$ when symbol Table has character to 3-digit number mapping.**

The value of $XML_N$ ((i.e., result of character translation)) in this case is given in Listing 4. The compression ratio corresponding to a three-digit symbol table is shown in Table 4. It is clear from the table that even with a 3-digit symbol table, the proposed method is superior to XMill and SMCA in compressing small size XML documents.

### E. COMPRESSION OF XML CONTAINING BINARY DATA

XML documents may also contain binary data. It is important, therefore, that the proposed technique for XML compression works on binary data. To denote the start of binary data in an XML document, we introduce the fourteenth symbol in $XML_N$ as 'B'. 'B' like other contents in $XML_N$ is also converted into a unique 4-bit binary in $XML_S$. To convert binary data to $XML_N$, we apply a technique like Base-64 [26]. This involves dividing the binary data into 6-bit chunks. If the two-digit number that a 6-bit chunk represents is N, then the number N+1 is calculated and the two digits of N+1 are appended to $XML_N$. While creating groups of binary data of size 6 bits, if the last group cannot be made of 6 bits owing to very few bits remaining, then extra 0s are added to make it 6 bits long. Binary data is always represented in terms of a number of bytes. Let us assume, for example, that the binary data here is 3 bytes long, that is, 24 bits. We make four groups of 6 bits from these 24 bits. A group of 6 bits represents a number having two decimal digits. In our approach, we use a four-bit binary (i.e., a nibble) to represent each decimal digit in the packing stage of compression. Two decimal digits obtained from a group of 6 bits will, therefore, be accommodated in 8 bits. This means that it would take 8 bits in $XML_S$ to represent binary data 6 bits long and thus it will take 32 bits to represent binary data of 24 bits. The number of bytes required in $XML_S$ to represent the binary data therefore becomes 4/3 times the actual size of the binary data. This expansion is mediated by the back-end compression. Representing binary data with the proposed method permits 10 symbols in a nibble. Therefore, only 100 different values are possible in a byte instead of 256 values. Being able to accommodate only 100 values in a byte increases the probability of repetition of a byte. Repeating bytes in a binary document are very well handled by back-end compressors like PPMD [3]. Since previous research does not consider compression of binary data, we have not been able to conduct a comparative study of our technique in this regard.

## IV. EVALUATION

Experiments are conducted in this section to evaluate the hypotheses raised in Section I. In the experiments conducted,

we demonstrate that better compression is achieved if fewer containers are used to compress an XML. The experiments also show that in large size XML documents with a relatively small number of containers, the number of containers does not influence the compression ratio. The experiments demonstrate that, given the independence of the proposed technique from the use of containers for compression, the proposed technique performs very well on compression of small size web API XMLs.

### A. EXPERIMENTAL DESIGN AND DATASET

To assess the efficacy of the proposed technique, we apply it to two benchmark datasets: the first is the one used in SMCA [6] (Section IV-B); and the second is the one used in the survey paper of Sakr [1] (section IV-C). The proposed compression technique is implemented using Python3. It is important to note that in the experiments, we calculate the compression ratio as the fraction of the size of the original XML document over the size of the compressed document ($CompressionRatio(CR) = XMLSize/CompressedSize$); a higher compression ratio, therefore, implies a better compression. The configuration of the system used for the experiments comprises: an i3-4005U CPU @ 1.70 GHz processor and a 4 GB DDR4 RAM.

#### 1) THE FIRST BENCHMARK DATASET

The first benchmark that we compare our technique against is the SMCA [6] method. We choose SMCA because it is the leading XML compression technique in terms of compression ratio. We were unable to find an implementation of SMCA and therefore compared the compression ratios achieved for our proposed technique with the ratios reported in SMCA [6]. The results are included in Section IV-B. The dataset used to compute the compression ratios of our technique is the same used by SMCA.

The dataset used by SMCA [6] comprises 160 XML documents, 1.xml, 2.xml, and so on, divided into four groups of 40 XMLs each, labelled ''small'', ''medium'', ''large'', and ''very large'', to indicate four different size ranges. The XML documents in each of these groups are then aggregated to form four new XML documents, i.e., small.xml, medium.xml, large.xml, and vlarge.xml. In addition, 16 more XML documents, S1.xml to S16.xml, are created by combining the first 10, 20, 30 and 40 XMLs of each of the above groups; i.e., S1.xml combines the first 10 documents of the small set, 1.xml to 10.xml. The XMLs S5.xml to S8.xml are created by combining all 40 XMLs of the small group and respectively the first 10, 20, 30 and 40 XMLs of the medium group (for example: s5.xml is a combination of the 40 XML documents of the small group and the first 10 documents of the medium group; s6.xml is a combination of the 40 XML documents of the small group and the first 20 XML documents of the medium group and so on until s8.xml). Similarly, S9.xml to S12.xml are combinations of all the XMLs of the small and medium groups and respectively the first 10, 20, 30 and 40 XMLs of the large group. Finally, S13.xml to S16.xml are combinations of all the XMLs of the small, medium, and large groups and respectively the first 10, 20, 30 and 40 XMLs of the vlarge group. Therefore, in this way the first dataset contains a total of 180 XMLs. The original 160 documents, small.xml to vlarge.xml and S1.xml to S16.xml.

As Table 4 shows, the dataset contains XML documents ranging from 4673 bytes to 2072248 bytes. The number of containers in each of these XML documents depends on the number of independent unique paths in each document. This number (18, in this dataset) is the same in all XML documents in spite of large variations in terms of amount of data. For a small XML document, 18 containers have a negative impact on compression because each container has very little data. Therefore, the compression results of SMCA which utilizes containers for compression is inferior for small XML documents. The proposed technique, on the other hand, only uses one container (i.e., $XML_S$) for compression and thus does much better than SMCA with small XML documents. As the size of the XML increases, the data content in the 18 containers used by SMCA starts to increase, and so does the compression ratio. The 18 containers soon become insignificant when the size of the XML becomes very large. Conversely, the proposed technique continues to use one container for large XML documents and its compression ratio starts to converge with that of SMCA. Figure 3 shows a comparison of compression ratios of the proposed compression technique with SMCA. The proposed technique maps frequently appearing XML tags with numbers having a small number of digits in the tag table. This further improves the compression as tags appearing a large number of times are represented by short numbers that take less space. In this way the Hypotheses raised in Section I are verified.

#### 2) THE SECOND BENCHMARK DATASET

The second benchmark dataset comprises 24 XML documents, three of which could not be downloaded. We use 21 of these documents, therefore, and the range in size from 533,579 to 137,538,931 bytes. This dataset was used to evaluate an earlier generation of XML compressors [1]. We choose the XMill XML compression algorithm from this generation of compression algorithm. This is because XMill is widely considered to be a standard XML compression technique and consistently maintains a good compression ratio. Furthermore, the implementation of XMill is easily available. We downloaded the XMill implementation from the following url: *https://sourceforge.net/projects/xmill/* and compared the compression ratios and run-times of both methods. The results of these experiments are reported in Section IV-C. Figure 6 shows a comparison of the proposed method and XMill on the second dataset.

### B. COMPRESSING API-RESPONSE XMLs

Certain compression techniques are specifically designed for compression of XML documents, produced and consumed by SOAP-based web services. Today, the state-of-the-art in such

techniques is SMCA [6]. The paper demonstrates the efficacy of SMCA using a dataset comprising SOAP requests and responses from various sources. These requests and responses are systematically aggregated to form documents of varying sizes. The dataset is now considered a standard and several compression algorithms have been tested against it, e.g., [15] and [16], with SMCA leading in terms of compression ratio.

According to [6], the SMCA technique was implemented using Visual Basic on an Intel(R) Xeon(R) CPU E5-1630 v3@ 3.70 GHz, 16 GB RAM. Tables 4 and 5 include data on the performance of SMCA provided in [6]. The same tables include results on the performance of our technique and these are based on actual execution on our system using this dataset. The other significant compression technique XMill, as stated earlier, was downloaded by us on our system (configuration of our system is provided in the previous section) using the executable file downloaded from the sourceforge website and made to work on the same dataset.

**TABLE 4.** Compression ratio comparison on SMCA dataset.

| XML file | Size (bytes) | Our method CR (2 Digit Symbol Table) (Only English) | Our method CR (3 Digit Symbol Table)(Non-English) | SMCA CR | XMill CR |
|---|---|---|---|---|---|
| S1 | 4673 | 6.49 | 5.45 | 4.07 | 4.69 |
| S2 | 8567 | 8.61 | 7.40 | 5.45 | 6.38 |
| S3 | 12147 | 9.72 | 8.53 | 6.24 | 7.22 |
| S4 | 16475 | 10.84 | 9.68 | 6.93 | 8.02 |
| S5 | 33563 | 13.59 | 12.41 | 9.39 | 10.04 |
| S6 | 50716 | 15.09 | 13.79 | 10.79 | 10.06 |
| S7 | 71528 | 16.33 | 15.00 | 12.13 | 9.68 |
| S8 | 90785 | 17.13 | 15.78 | 13.20 | 9.06 |
| S9 | 211795 | 19.61 | 18.05 | 17.04 | 10.41 |
| S10 | 336198 | 20.63 | 19.01 | 18.85 | 10.97 |
| S11 | 456942 | 21.15 | 19.47 | 19.91 | 11.36 |
| S12 | 555800 | 21.54 | 19.82 | 20.56 | 11.42 |
| S13 | 939150 | 22.31 | 20.54 | 21.96 | 12.46 |
| S14 | 1339834 | 22.75 | 20.88 | 22.59 | 13.08 |
| S15 | 1717316 | 22.96 | 21.05 | 22.84 | 13.43 |
| S16 | 2072248 | 23.15 | 21.20 | 23.10 | 13.69 |
| small | 16475 | 10.87 | 9.68 | 6.94 | 8.02 |
| medium | 74311 | 16.54 | 15.20 | 13.03 | 8.59 |
| large | 465016 | 21.45 | 19.38 | 20.66 | 11.31 |
| very large | 1516449 | 22.85 | 20.79 | 23.36 | 13.26 |

**TABLE 5.** Time comparison (ms).

| XML file | XML Size(bytes) | Our method | SMCA | XMill |
|---|---|---|---|---|
| S1 | 4673 | 71 | 13 | 27 |
| S2 | 8567 | 70 | 15 | 25 |
| S3 | 12147 | 80 | 20 | 24 |
| S4 | 16475 | 71 | 27 | 25 |
| S5 | 33563 | 90 | 45 | 26 |
| S6 | 50716 | 130 | 62 | 28 |
| S7 | 71528 | 140 | 76 | 33 |
| S8 | 90785 | 142 | 78 | 32 |
| S9 | 211795 | 150 | 91 | 43 |
| S10 | 336198 | 226 | 138 | 45 |
| S11 | 456942 | 240 | 187 | 51 |
| S12 | 555800 | 312 | 219 | 58 |
| S13 | 939150 | 420 | 311 | 70 |
| S14 | 1339834 | 688 | 405 | 90 |
| S15 | 1717316 | 812 | 500 | 102 |
| S16 | 2072248 | 1013 | 655 | 125 |

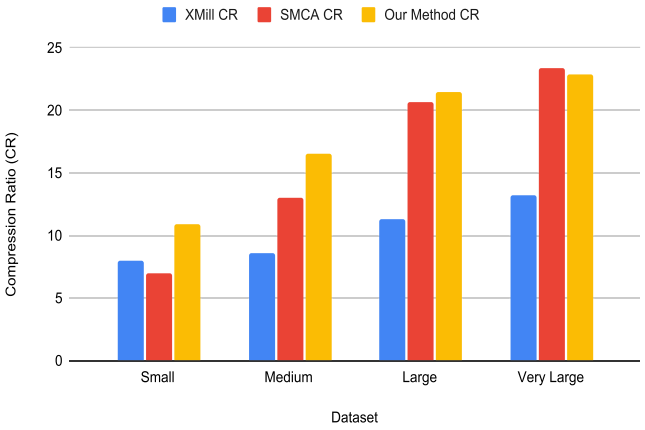


**FIGURE 2.** Compression ratio comparison on small, Medium, Large and very large group (SMCA dataset).

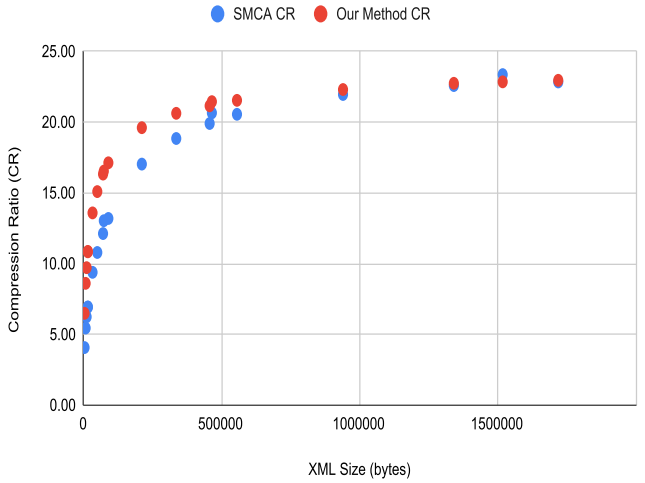


**FIGURE 3.** (SMCA vs Proposed Method) compression ratio comparison on 20 XMLs of SMCA dataset.

Table 4 provides data on the compression ratio of our technique and that of SMCA and XMill. The same table is graphically illustrated in the Figures 3 and 2. The compression ratios of the three techniques are compared on 20 XML documents from the dataset namely small.xml, medium.xml, large.xml, vlarge.xml and XMLs from S1.xml to S16.xml. The compression ratio results clearly indicate that the proposed technique outperforms both SMCA and XMill for XML documents of all sizes except very large ones where our compression ratio is somewhat equal to that of SMCA. Our compression ratio is much superior when compared to SMCA for small size XML documents but the gap progressively reduces as the size of the XML document increases. SMCA mostly outperforms XMill except in the case of very small XML documents.

In web services the XML documents exchanged are mostly small in size and thus the proposed technique is the most appropriate. SMCA, on the other hand, does not perform well with small XML documents and hence is not suitable to be

used with web-services. As both SMCA and XMill use text compressors at the back-end, we compare their compression results with $XML_B$.

In terms of the time required for compression, Table 5 shows that our technique takes longer than SMCA and XMill. This is mainly because our technique supports parsing and querying even in the compressed form (more precisely, parsing and querying are supported until the stage that $XML_S$ is formed as described in the relevant section earlier). SMCA, on the other hand, forms compressed documents that do not support parsing and querying and hence the shorter time for compression.

Comparison of the compression ratio of our proposed technique with that of XMill is shown in Figure 4. The graphs have been traced over all the 180 XML document of the dataset. The proposed technique achieves a much better compression ratio across the dataset. Most XML documents in the dataset are smaller than 60000 bytes and therefore for better visualisation and more detailed analysis we compare the two techniques for XML documents smaller than 60000 bytes in Figure 5. Such documents make up 168 of the total 180 XML documents in the dataset and our proposed technique is seen to perform much better than XMill.

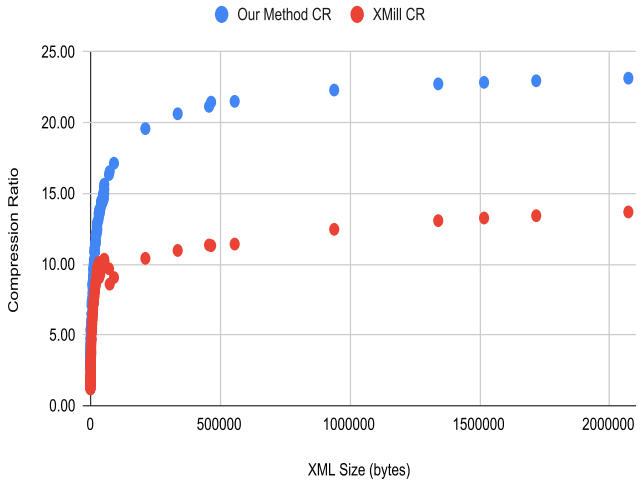


**FIGURE 4.** (XMill vs Proposed Method) compression ratio comparison on all 180 XMLs of SMCA dataset.

### C. COMPRESSING DOCUMENT-STYLE XMLs

The 24 XMLs used by [1] exist in two forms each, the original documents and the structured documents. Original XML documents are full XML documents comprising both structure and data parts. Structured XML documents, on the other hand, consist of only opening and closing tags. These documents, therefore, only have structure information and no data part.

We compare the compression ratios of the proposed technique with that of XMill using only the original XML documents, the ones with both structure and data parts. The results are shown in Figure 6. It is clear that our technique outperforms XMill in terms of compression ratio with every

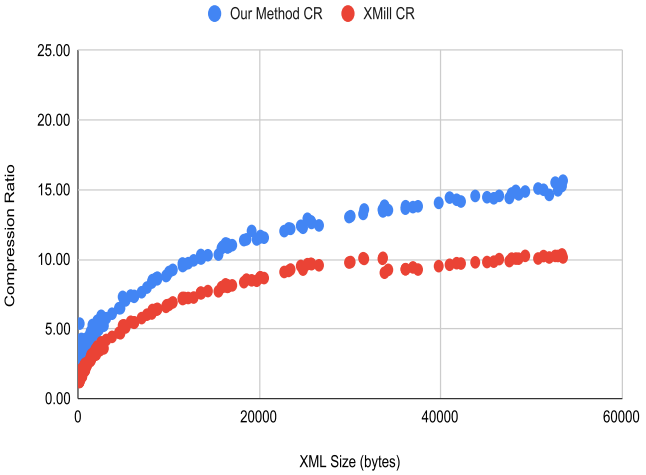


**FIGURE 5.** (XMill vs Proposed Method) compression ratio comparison on 168 XMLs having sizes less than 60000 bytes using SMCA dataset.

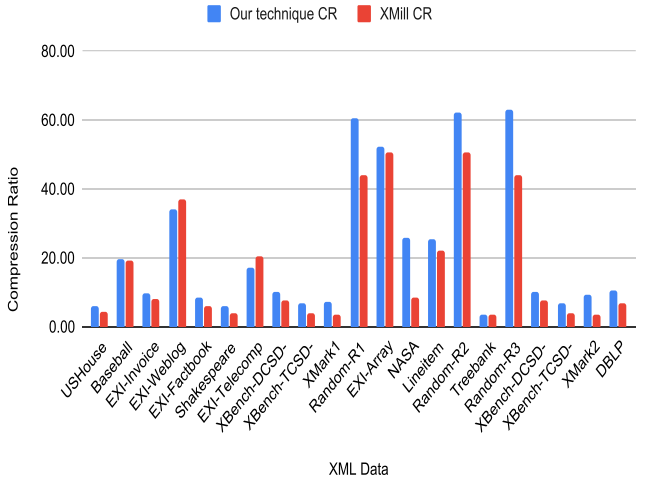


**FIGURE 6.** Compression ratio comparison on original XML documents (longer the bar better is the compression ratio).

XML document except two. These two documents have a large number of tags with relatively small sized free text data between the tags.

### D. EVALUATING THE QUERYABLE FEATURE OF THE PROPOSED TECHNIQUE

The proposed technique maintains the characteristic of being queryable up to the stage of $XML_S$ (the various stages of compression are shown in Figure 1). After applying back-end compression, however, the compressed document is no longer queryable. The compression ratios for documents at the stage of forming $XML_S$ are naturally smaller than the final compression ratios which also incorporate back-end compression. The significance of the documents at this stage, as stated earlier, is that they are queryable as opposed to the document formed after back-end compression.

XGrind is perhaps the oldest and most significant research endeavour in the direction of Queryable XML Compressors,

**TABLE 6.** Compression Ratio of $XML_S$ (proposed method) vs QRFXFreeze.

| XML file | XML Size (MB) | $XML_S$ CR | QRFXFreeze |
|---|---|---|---|
| XMark | 118 | 2.45 | 2.38 |
| DBLP | 138 | 1.91 | 1.92 |
| TreeBank | 89 | 2.20 | 1.17 |
| Shakespeare | 7.9 | 2.38 | 2.22 |
| SwissPort | 109 | 2.18 | 2.13 |

whereas QRFXFreeze [24] is a more recent technique and outperforms most queryable compressors in terms of compression ratio. The compression ratio of the proposed technique, at the stage of $XML_S$, is compared with the compression ratio of QRFXFreeze. The dataset that is used to compare QRFXFreeze with other queryable compression algorithms in [24] is also used here to compare the proposed technique with QRFXFreeze. Table 6 shows this comparison and $XML_S$ of the proposed technique has a better compression ratio than QRFXFreeze for four out of the five XML documents used. In [24] the compression ratio is calculated using the formula

$$cr = [1 - \frac{sizeof\,(compressedfile)}{sizeof\,(originalfile)}] * 100$$

. To make this consistent with our technique we calculated

$$\frac{sizeof\,(originalfile)}{sizeof\,(compressedfile)} = \frac{100}{(100 - cr)}$$

. Further

$$\frac{sizeof\,(originalfile)}{sizeof\,(compressedfile)}$$

is compared with the compression ratio of the proposed technique.

### E. EVALUATION OF JSON COMPRESSION

A JSON document also contains a data part and a structure part as discussed in Section III-C. Therefore, the same outputs, $XML_N$ (subsequently $XML_S$ and $XML_B$), are generated irrespective of whether XML and JSON representation is used. However, as major research endeavours use XML compression, we have also evaluated our research using XML.

### F. SIZE, INFORMATION AND COMPRESSION ANALYSIS

XML documents may vary in their nature based on the number of characters dedicated to their data part and the number of characters dedicated to their structure part. The structure part of an XML comprises tags and attributes whereas the data part is the content between tags. An XML document may have a large data part with a large number of characters and very few tags and attributes. Another document may have a large number of tags and attributes with a large number of characters dedicated to these and very few characters for data. Certain XML documents may have a small number of tags and attributes but the size of each tag and/or attribute may be very large and hence a large number of characters dedicated to the same.

Our proposed technique for XML compression varies in its compression efficacy with different types of XML. In this section, we seek to understand the behaviour of the technique in this respect. To this end, we create synthetic XML documents of varying characteristics as described above. We start with a template XML document consisting of $n$ slots. A slot can be filled with any one of the following elements of an XML document: an opening tag, a closing tag, an attribute name, a single character of an attribute value, or a single character of data content. It is important to note that irrespective of the size of an opening tag, closing tag, and attribute value, each occupies a single slot; whereas one character of the data occupies a single slot. This means that if a single element of data comprises 10 characters, 10 slots would be required to accommodate the data element. Conversely, if an opening tag, closing tag, or attribute name is of size 10 characters, it would be accommodated in just 1 slot. The intent of defining slots in this manner is to understand the varying nature of XML documents and the variation in the effectiveness of the proposed technique with such varying documents.

We start with a document comprising only opening and closing tags that fill all the slots, resulting in an XML of *n/2* opening and *n/2* closing tags. The next XML document is created by substituting one pair of opening and closing tags with two data characters (as stated earlier, one slot can accommodate an entire opening or closing tag but only one character of data). Next another pair of opening and closing tags is substituted with data characters. This is done with successive pairs of opening and closing tags being replaced with data characters until *n/2* XML documents are created with the first one comprising only tags and *n/2* empty elements, and the last one containing a single element (just one pair of opening and closing tags) and *n-2* data characters.

To understand the behaviour of the proposed technique, we express the nature of the XML documents in terms of *Information*. To do this, let us assume that an XML document with $n$ slots has a total of $t$ tags with an average size of $x$ characters per tag, a total of $a$ attributes with an average size of $y$ characters per attribute, and has a total of $d$ characters in the data part of the XML. Following this assumption, the total size of an XML with $n$ slots is $(t * x + a * y + d)$ and $2t + a + d = n$ as it is assumed that the $n$ slots are filled with the given number of tags, attributes, and data characters. The ratio $(t + a + d)/(t * x + a * y + d)$ represents the *information* that the XML carries and this ranges between *0* and *1*. Among XML documents with $n$ slots, the XML that has a smaller value of $(t * x + a * y + d)$, carries more *information* and we notice that such documents (that carry more information) perform poorly in terms of compression ratio with the proposed compression technique. When $t$ and $a$ are both 0, all the slots in the XML document are filled with single characters of data and exhibits the maximum *information*, i.e., 1. With such documents, the proposed technique returns the worst compression ratio. Similarly when $d = 0$ and $t \ll 2t * x$, $a \ll a * y$, then all the $n$ slots are filled with large tags or attribute names and the *information* that the XML document

carries is small, i.e., close to zero. In such cases we notice that the proposed technique gives good compression ratios.

We seek to understand the behaviour of the proposed technique in a little more detail by studying the variation of its compression ratio with varying *information*. To do this, three sets of XML documents are created each with the number of slots *n* being set at 10000. Each set comprises 5000 XML documents created in a manner described earlier: the first document in each set comprises only opening and closing tags; in a document with 10000 slots, this works out to 5000 of each opening and closing tags. For successive XML documents, tags are replaced with data characters and the next document has 4999 opening and closing tags and 2 characters of data. This is continued until the last (5000th) XML document in each set has just 1 opening and closing tag and 9998 data characters.

Each set of XML documents created through this process is populated with tags from three different schemas. The first set of documents is created using the ''XBench-TCSD-Normal_Structural.xml'' document from the survey paper [1] and comprises very small sized tags of average length 2 characters per tag. The second set is created using the ''S16.xml'' document used with SMCA [6] that comprises medium-size tags. The average length of tags in ''S16.xml'' is 10 characters per tags. Finally the third set uses the ''EXI-factbook_Structural.xml'' document again from the survey paper [1] that comprises large tags with an average size of 24.56 characters per tag.

Figure 7 demonstrates the behaviour of the proposed compression technique. For the same value of *information*, an XML with a large tag size results in a better compression ratio. The compression ratios for the XML documents of all three sets decrease with an increase in the value of *information*. This is very apparent for the sets with large and medium sized tags and less apparent for the documents with the set of documents with small tags. The same relation holds true for small tag documents but the graph appears like a straight line because of the small compression ratios. In Figure 7, a surge

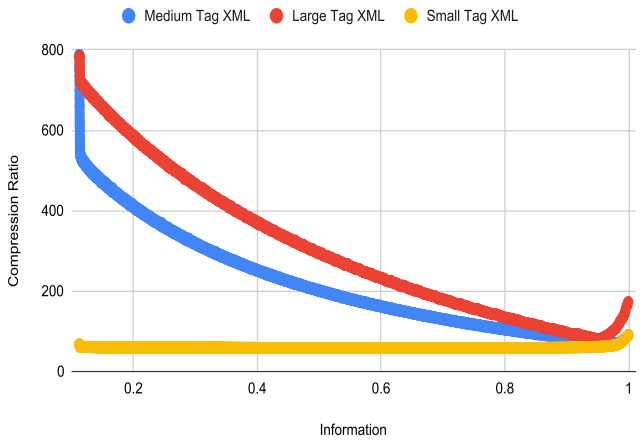


**FIGURE 7. Behaviour of the proposed technique on various types of similar XMLs (5000 XMLs in each set).**

appears in the compression ratio for documents whose value of *Information* is close to 1. This is because XML documents are created for our experiments by successively reducing the number of tags one at a time and replacing these and thus increasing the data content of the document one character at a time. When the number of tags is sufficiently small and the data content sufficiently large, a further decrease in number of tags and their replacement with data characters does not significantly reduce the size of the new XML, while resulting in comparable improvement in compression. This results in what appears to be a little like a surge in the compression ratio.

### G. COMPLEXITY ANALYSIS

Decompression of compressed document is the reverse of compression and involves the following procedure: the back-end decompressor is first used on the $XML_B$ document to get $XML_S$; next the nibbles from $XML_S$ are extracted one-by-one to get $XML_N$; and finally, the symbol and tag tables are used to get the original *XML*. The complexity of both the compression and decompression processes depends on: the number of slots in the *XML* document, and the complexity of the PPM algorithm, O(n) where n is the number of bytes in $XML_S$. Slots are the substitutable entities of an XML that are substituted using symbol and tag tables. These include tag names, attribute names, a character in data, and others. The slots of an *XML* are more comprehensively described and formally defined in Section IV-F. Both compression and decompression involve substitution followed by PPM compression or decompression. The number of slots is the same both in plain XML and in $XML_S$. Therefore, the complexity of both the compression and decompression processes is O(s+n).

### H. DISCUSSION

We compare the efficacy of our method against state-of-the-art competitors like XMill [4], QRFXFreeze [24], and the recent SMCA [6]. XMill is the pioneer among XML compression techniques and continues to serve as a benchmark. SMCA is the latest and the state of the art in XML compression. The Figure labeled ''(SMCA vs Proposed Method) Compression Ratio Comparison on 20 XMLs of SMCA Dataset'' i.e. the Figure 3 clearly indicates that our method is superior to SMCA in terms of compression ratio for small and medium size documents. For large documents, however, the proposed method performs similarly to SMCA, and in some cases SMCA performs marginally better. Figures 2 and 5 show that our method performs better than XMill on documents of all sizes. QRFXFreeze, unlike SMCA and XMill, is a queryable XML compressor. The proposed method is also queryable until the final back-end compression stage and hence we compare the queryable form (which we called $XML_S$, shown in Figure 1 of the proposed method with QRFXFreeze. QRFXFreeze has only been tested on XML documents of very large size and we, therefore, compare $XML_S$ with QRFXFreeze for such documents. We notice that

$XML_S$ comfortably outperforms QRFXFreeze in terms of compression ratio as shown in Figure or Table 6.

Note that SMCA and XMill, as stated earlier, are non-queryable and hence we compare their outputs with $XML_B$, the final, non-queryable, output of our method, as produced by the last stage of back-end compression is done.

In Section IV-F, we try to gain some insight on the kinds of documents that are most amenable to compression and hence provide superior compression ratios. We find that our method gives inferior compression ratios for XML documents with a relatively large number of characters in the data part, as compared to the structure part. On the other hand, if the structure part of the XML documents has a large number of characters our method works better.

It is important to remember that JSON documents, like XML documents, are divided into structure and data parts, hence the proposed method is equally applicable and effective on JSON documents as well. Recently, a new encoding method, the Google Protocol Buffer [25], was introduced. This can be used to represent messages in a format that is shorter than JSON. In Google protocol buffer messages are represented in a binary form, in terms of key-value pairs of structured and free-text data. Different types of data are represented using different encoding schemes in Google protocol buffer. Although the Google Protocol Buffer is an interesting option for resource representation, but it is not as popular and is seldom used in web services. Messages between web services are still mostly transferred as JSON or XML.

## V. LIMITATIONS

Although the proposed method performs better for small XMLs of web-based APIs, it has some limitations.

- As the number of characters in the XML increases, the digit count in the numbers used in the symbol tables increases. Therefore, the effectiveness of the proposed method in cases where a large number of characters is required is a little suspect. An example of this is its use for cases where non-English characters need to be used in the data part of the XML. This would make the compression less effective.
- When binary data is compressed with the proposed method, the size of the binary content in the intermediate XMLS increases by 1.33 times instead of decreasing. The back-end compression partially compensates for this expansion.
- The compression ratio of the proposed method on small XML documents is better than SMCA. However, in case of large XML documents, especially those larger than 2MB, the compression ratio of the proposed method is mostly equal to that of SMCA.

## VI. CONCLUSION AND FUTURE WORK

The key contribution of this work is a novel compression technique for structured documents like XML and JSON. The design is motivated by the need to improve the efficiency of network usage during service message exchange.

The proposed technique takes advantage of the structure of XML documents and relies on the intuition that the natural language underlying the messages exchanged requires fewer characters than the complete ASCII character set, under the realistic assumption that most service APIs are based on the English language. This, therefore, enables us to map structure tags and characters of the data elements into a much smaller character set. Beyond this translation step, the proposed technique further reduces the size of the document through a byte-packing step. This results in a significantly compressed document that importantly retains its queryable characteristic. The size of the document may further be reduced using a traditional compression algorithm but at this stage the document loses the property of being queryable.

Our experiments demonstrate that our method comfortably outperforms the current state-of-the-art compression technique, SMCA, in terms of compression ratio on files smaller than 1 MB (Table 4). On files larger than 1 MB, the proposed technique returns a compression ratio on par with that of SMCA. It is important to note that, for the most part, XML messages generated during interactions between ReST services are quite small. The proposed technique, therefore, is quite effective in improving the utilization of network resources in Service-Oriented systems which are mostly ReSTful in nature today, as compared to SMCA.

For document-style XML, the proposed technique outperforms XMill, long considered a standard compression technique for such XML documents, in terms of compression ratio (Figure 6). In this case, however, the final form of the compressed document using the proposed technique is non-queryable. This is because the final stage of compression involves utilising a traditional text-compression algorithm.

The stage of compression just before this final stage (as shown in Figure 1) results in a document that is queryable. The compression ratios for these queryable compressed documents are compared with the compression ratios of the state-of-the-art query-friendly compressor XQRFFreeze [24]. The results (Table 6) show that the proposed technique compression ratios at the queryable stage outperforms the best in literature.

In the future, we will attempt to address the limitations mentioned above. Better compression of the binary data being transferred through XML or JSON can be an important avenue for future research. In addition, the compression of XML or JSON documents using non-English characters can be improved.

## REFERENCES

[1] S. Sakr, "XML compression techniques: A survey and comparison," *J. Comput. Syst. Sci.*, vol. 75, no. 5, pp. 303–322, Aug. 2009.
[2] J. Seward. (1996). *Bzip2 and Libbzip2*. [Online]. Available: http://www.bzip.org
[3] J. Cleary and I. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun.*, vol. COM-32, no. 4, pp. 396–402, Apr. 1984.
[4] H. Liefke and D. Suciu, "XMill: An efficient compressor for XML data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2000, pp. 153–164.

[5] P. M. Tolani and J. R. Haritsa, "XGrind: A query-friendly XML compressor," in *Proc. 18th Int. Conf. Data Eng.*, Feb. 2002, pp. 225–234.
[6] N. Haroune-Belkacem, F. Semchedine, A. Al-Shammari, and D. Aissani, "SMCA: An efficient SOAP messages compression and aggregation technique for improving Web services performance," *J. Parallel Distrib. Comput.*, vol. 133, pp. 149–158, Nov. 2019.
[7] *W3Techs—World Wide Web Technology Surveys*. Accessed: Mar. 13, 2021. [Online]. Available: https://w3techs.com/technologies/overview/content_language
[8] M. Girardot and N. Sundaresan, "Millau: An encoding format for efficient representation and exchange of XML over the Web," *Comput. Netw.*, vol. 33, nos. 1–6, pp. 747–765, Jun. 2000.
[9] G. Busatto, M. Lohrey, and S. Maneth, "Efficient memory representation of XML document trees," *Inf. Syst.*, vol. 33, nos. 4–5, pp. 456–474, Jun. 2008.
[10] M. Lohrey, S. Maneth, and R. Mennicke, "XML tree structure compression using RePair," *Inf. Syst.*, vol. 38, no. 8, pp. 1150–1167, Nov. 2013.
[11] M. Bousquet-Mélou, M. Lohrey, S. Maneth, and E. Noeth, "XML compression via directed acyclic graphs," *Theory Comput. Syst.*, vol. 57, no. 4, pp. 1322–1371, Nov. 2015.
[12] W. Li, "Xcomp: An XML compression tool," Doctoral dissertation, School Comput. Sci., Univ. Waterloo, Waterloo, ON, Canada, 2003.
[13] J.-K. Min, M.-J. Park, and C.-W. Chung, "XPRESS: A queriable compression for XML data," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2003, pp. 122–133.
[14] P. Skibiński and J. Swacha, "Combining efficient XML compression with query processing," in *Proc. East Eur. Conf. Adv. Databases Inf. Syst.*, Berlin, Germany: Springer, Sep. 2007, pp. 330–342.
[15] D. Al-Shammary and I. Khalil, "SOAP Web services compression using variable and fixed length coding," in *Proc. 9th IEEE Int. Symp. Netw. Comput. Appl.*, Jul. 2010, pp. 84–91.
[16] A. M. Abbas, A. A. Bakar, and M. Z. Ahmad, "Fast dynamic clustering SOAP messages based compression and aggregation model for enhanced performance of Web services," *J. Netw. Comput. Appl.*, vol. 41, pp. 80–88, May 2014.
[17] C. C. Aggarwal, N. Ta, J. Wang, J. Feng, and M. Zaki, "Xproj: A framework for projected structural clustering of xml documents," in *Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2007, pp. 46–55.
[18] A. Al-Shammari, C. Liu, M. Naseriparsa, B. Q. Vo, T. Anwar, and R. Zhou, "A framework for clustering and dynamic maintenance of XML documents," in *Proc. Int. Conf. Adv. Data Mining Appl.* Cham, Switzerland: Springer, Nov. 2017, pp. 399–412.
[19] D. Al-Shammary and I. Khalil, "Dynamic fractal clustering technique for SOAP Web messages," in *Proc. IEEE Int. Conf. Services Comput.*, Jul. 2011, pp. 96–103.
[20] D. Al-Shammary and I. Khalil, "Redundancy-aware SOAP messages compression and aggregation for enhanced performance," *J. Netw. Comput. Appl.*, vol. 35, no. 1, pp. 365–381, Jan. 2012.
[21] D. Al-Shammary, I. Khalil, and Z. Tari, "A distributed aggregation and fast fractal clustering approach for SOAP traffic," *J. Netw. Comput. Appl.*, vol. 41, pp. 1–14, May 2014.
[22] D. Al-Shammary, I. Khalil, Z. Tari, and A. Y. Zomaya, "Fractal self-similarity measurements based clustering technique for SOAP Web messages," *J. Parallel Distrib. Comput.*, vol. 73, no. 5, pp. 664–676, May 2013.
[23] A. Algergawy, M. Mesiti, R. Nayak, and G. Saake, "XML data clustering: An overview," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 1–41, Oct. 2011.
[24] R. Senthilkumar, G. Nandagopal, and D. Ronald, "QRFXFreeze: Queryable compressor for RFX," *Sci. World J.*, vol. 2015, pp. 1–8, Jan. 2015.
[25] G. Kaur and M. M. Fuad, "An evaluation of protocol buffer," in *Proc. IEEE SoutheastCon (SoutheastCon)*, Mar. 2010, pp. 459–462.
[26] S. Josefsson, *The Base16, Base32, and Base64 Data Encodings*, document RFC 4648, Oct. 2006, pp. 1–18.

**GYAN P. TIWARY** received the B.Tech. degree in information technology from the Birsa Institute of Technology Sindri (BIT Sindri), Dhanbad, India, in 2010, and the M.Tech. degree in computer applications from the IIT (Indian School of Mines) Dhanbad, India, in 2013. He is currently pursuing the Ph.D. degree in computer science and engineering with IIT Indore, India.

From 2019 to 2020, he was awarded with the Overseas Visiting Doctoral Fellowship by Science and Engineering Research Board, Government of India, to work as a Visiting Research Scholar at the University of Alberta, Canada. His research interests include service oriented architecture, privacy and security accepts of web services, document compression, and cryptography.

Mr. Tiwary received fellowship and grants include the Overseas Visiting Doctoral Fellowship (Science and Engineering Research Board), the Visvesvaraya Ph.D. Scheme Fellowship during Ph.D. degree at IIT Indore, and the GATE Fellowship during the study of M.Tech. degree at IIT (ISM) Dhanbad.

**ELENI STROULIA** (Member, IEEE) is currently a Professor with the Department of Computing Science, University of Alberta. From 2011 to 2016, she held the NSERC/AITF Industrial Research Chair on service systems management with IBM. Her Flagship Project in the area of health care is the Smart Condo in which she investigates the use of technology to support people with chronic conditions live independently longer and to educate health-science students to provide better care for these clients. She has played leadership roles in the GRAND and AGE-WELL NCEs. She has supervised more than 60 graduate students and PDFs, who have gone forward to stellar academic and industrial careers. Her research interests include addressing industry-driven problems, adopting AI, and machine-learning methods to improve or automate tasks. Since January 2020, she has been the Director of the AI4Society Signature Area. In 2018, she received the McCalla Professorship. In 2019, she was recognized with the Killam Award for Excellence in Mentoring. In 2011, Smart-Condo Team received the UofA Teaching Unit Award.

**ABHISHEK SRIVASTAVA** received the Ph.D. degree from the University of Alberta, Canada, in 2011. He is currently an Associate Professor with the Discipline of Computer Science and Engineering, IIT Indore. His group at IIT Indore has been involved in research on service-oriented systems most commonly realized through web-services. More recently, the group has been interested in applying these ideas in the realm of the Internet of Things. The ideas explored include coming up with technology agnostic solutions for seamlessly linking the heterogeneous IoT deployments across domains. The group is also delving into utilizing machine learning adapted for constrained environments to effectively make sense of the huge amounts of data that emanate from the vast network of the IoT deployments.