# Compacting XML Data

Shuohao Zhang, Curtis Dyreson, and Zhe Dang

P. O. Box 642752,
Washington State University,
Pullman, WA 99164-2752, USA
{szhang2, cdyreson, zdang}@eecs.wsu.edu

**Abstract.** *Compression* aims to reduce the size of data without loss of information. *Compaction* is a special kind of compression in which the output is in the same language as the input. Compaction of an XML data forest produces a smaller XML forest, without losing any data. This paper develops a formal framework for the compaction of XML data and presents two compaction techniques.

## 1 Introduction

*Compression* aims to reduce the size of data without loss of information. It is useful because smaller data can save storage space and also network bandwidth when data is transmitted. A *compressor* generates a file smaller in size than the original; feeding this file to a *de-compressor* can recover the original.

XML is rapidly becoming a dominant media for data exchange over the Internet. Because XML data is usually quite verbose, compression is an important issue for XML. Several tools are already available for XML compression. One can use either a general purpose compressor such as gzip, or an XML-specific compressor (such as XMill [2]) to compress XML data.

This paper addresses a special kind of compression, called compaction, where the compressed output remains as XML. Existing compression techniques do not compact the data because they all produce a compressed file in a non-XML format, which only a special-purpose de-compressor can understand. The main benefit of compaction is that it is orthogonal to other compression techniques, so an XML file can be compacted and then compressed.

The general idea behind compaction is that the same data can be represented in XML in (several) different structures. Consider two XML data documents, author.xml and pub.xml, shown respectively in Fig. 1 and Fig. 2. The data is simple enough that we can rely on readers of the data to agree on its intended semantics. In author.xml author n1 writes a book t1, published by p1; author n2 is a co-author on book t1 and also independently writes a book t2, published by p2. pub.xml contains exactly the same information except that the structure is different. Both documents have data about the same two authors, the same two publishers, and the same two books. Each document similarly relates each book, author, and publisher, e.g., in both documents book t2 is authored by author n2 and published by publisher p2. Section 3 develops a formal framework that allows the implicit meaning or semantics of an XML data

M.L. Lee, K.L. Tan, and V. Wuwongse (Eds.): DASFAA 2006, LNCS 3882, pp. 767 – 776, 2006.

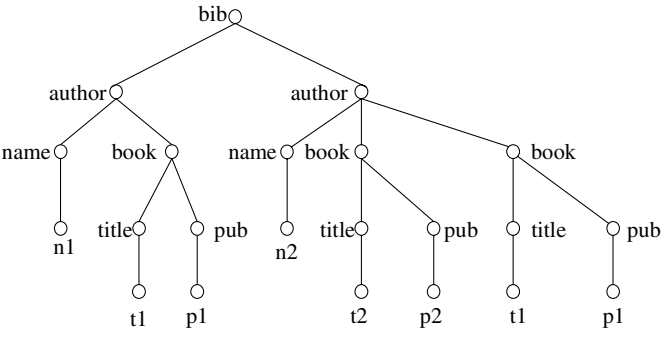


**Fig. 1.** author.xml

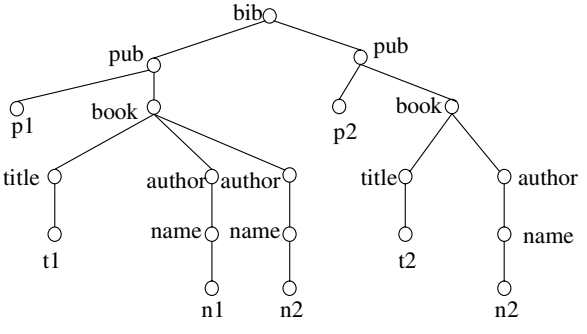


**Fig. 2.** pub.xml

collection to be determined and compared. For compaction, what interests us is the fact that author.xml and pub.xml have the same data but are of different sizes. Excluding text data, there are 14 elements in author.xml but only 13 elements in pub.xml. This suggests that author.xml could be compacted to at least the size of pub.xml. Of course, other compression techniques could potentially reduce the physical size of author.xml much further; however, only compacting produces output in XML.

Compaction is concerned with *logical* redundancy as much as *physical* redundancy. Note that in compaction we can measure the size of XML data by the number of elements. This differs from common compression tasks in which the size of a compressed file can only be measured by the disk space it occupies. While file size reflects the physical redundancy in a file, the number of *duplicate* nodes gives a better measure for redundancy on the logical level. By preserving the XML syntax in the output, compaction rearranges the original data to a new form with fewer places that are subject to update anomaly. Certainly, compacting an XML file may potentially (and usually does) compress the data at the same time. Though fewer elements does not guarantee a physically smaller file in general, it is usually so in practice.

This paper is organized as follows. Section 2 introduces preliminary concepts and Section 3 presents a semantics for XML that translates an XML data collection to a graph. Section 4 presents a compaction technique called *restructuring*, which

transform a forest to another without affecting the semantics. Section 5 presents related work, and Section 6 concludes the paper.

## 2 Preliminaries

This section defines preliminary concepts. We start with tree and forest.

**Definition** [tree]**.** A tree is a five tuple $(V, E, \Sigma, L, C)$, where $V$ is the *node* set, $E{:}V{\times}V$ is the *edge* set, $\Sigma$ is an *alphabet* of labels and text values, $L{:}V{\rightarrow}\Sigma$ is a *label function* that maps a node to its label, and $C{:}V{\rightarrow}\Sigma$ is a *value function* that maps a node to its *value*. ∎

This tree data model is different from the DOM data model [5]. It ignores sibling order and it does not model other kinds of DOM nodes such as attributes and comments. But the simpler model is sufficient for our purposes.

We often need to deal with an XML *data collection*, which is a group of XML documents or parts of XML documents. This can be modeled as a *forest* in general.

Labels can be used to partially identify nodes in a forest, but not to distinguish nodes of the same label. To further identify nodes of the same label, we need another characterization. One such characterization is a *type identifier*. Here we define nodes to be of the same *type* if they have the same label; *type identifier* is defined to be an identifier that identifies nodes of the same type. (Note that the term "type" is commonly used in the XML database literature but with varying meanings in different researches. In this paper, the type of a node is simply its label.) Such type identifiers observe the dependency among nodes of different types in a forest. For example, we may have the following dependencies in author.xml and pub.xml:

- an author depends on its corresponding name,
- a book depends on its corresponding title, and
- a name, title or publisher each depends on its value.

In each of these dependencies, one type is dependent on some other types or its own value. In a specific dependency, we call a node of the depending type a *depending* node; a depending node is identified by nodes or value corresponding to the deciding types, which we call the *identifying information* of the depending node. In general, we shall allow identifying information to be a combination of both nodes and values.

Usually, a node's identifying information is its immediate children (nodes or values). We further observe that, regardless of the relative position of the depending types and the deciding types, the identifying information is always "closest" to a dependent node. For example, if a book is identified by its title, then in the forest that title is closer to the book it identifies than it is to other books.

More precisely, suppose $v$ is a dependent node and $u$ is a type $t$ identifying node of $v$, then $u$ is closest in distance to $v$ among all type $t$ nodes. This observation suggests that we can employ this notion of *closeness* to locate the identifying information.

**Definition** [related nodes]**.** Let $v$ be a node of type $x$. Then $related(v, t) = \{x \mid x$ is a node of type $t$ and from among all the nodes of type $t$, $x$ is *closest* in distance to $v\}$. The distance between a pair of nodes is measured by the length of the path that connects the nodes. ∎

Using the notion of closest, related nodes, we formalize a type identifier as follows.

**Definition** [type identifier]**.** A type identifier $I$ of a type $t$ is a two-tuple ($I_{Type}$, $I_{Text}$), where $I_{Type} = \{x_1,\ldots, x_m\}$ and $I_{Text} = \{y_1,\ldots, y_n\}$ are each a set of types, $m$ and $n$ are non-negative integers and they are not both zero, and $t \notin I_{Type}$. Two type $t$ nodes $u$ and $v$ are *identical*, denoted $u \doteq v$, if and only if the following holds.

- When $m > 0$, for each $q$ in $related(v, x_i)$, $1 \le i \le m$, there exists a node $p$ in $related(u, x_i)$ such that $p \doteq q$; for each $p$ in $related(u, x_i)$, there exists a node $q$ in $related(v, x_i)$ such that $p \doteq q$.
- When $n > 0$, for each $q$ in $related(v, y_i)$, $1 \le i \le n$, there exists a node $p$ in $related(u, y_i)$ such that $C(p)=C(q)$; for each $q$ in $related(u, y_i)$, there exists a node $p$ in $related(v, y_i)$ such that $C(p)=C(q)$.

The following notation represents the dependency of type $t$ on the other types:

$$t \leftarrow x_1,\ldots, x_m; y_1,\ldots, y_n$$

where the delimiter symbol ";" is required, even if $m$ or $n$ is zero. ∎

The above definition recursively describes how a depending node is identified by a combination of nodes of other types and some values. The base case in the recursive definition is when the set $I_{Type}$ of type $t$ is empty. In this case, whether two type $t$ nodes $u$ and $v$ are identical is decided by comparing some values. If $I_{Type}$ is non-empty, then whether $u$ and $v$ are identical is recursively determined by whether nodes of some other types are identical. As a special case, $u$ and $v$ are identical if they are the same node. Using the type identifier notation, the dependencies in the motivating example are: 1) "author ← name;", 2) "book ← title;", 3) "name ← ; name", 4) "title ← ; title", and 5) "publisher ← ; publisher".

## 3 A Semantics for XML

We now illustrate a semantics for XML using the example in Section 1. We show a series of semantics-preserving operations that derives the semantics of a tree, which is a graph. As both author.xml and pub.xml are mapped to the same graph, they are regarded semantically equivalent.

First, we identify duplicate information, i.e., data that represents the same real-world entity. Duplicates are identified by the type identifiers. In author.xml, since the first and third book have the same title, and we have identifiers "book ← title;" and "title ← ; title", the fist and third book elements are duplicates.

Similarly, in pub.xml, the second and third author elements represent the same author, because of the identifiers "author ← name; " and "name ← ; name". The duplicate information is removed through a process called *node gluing*. Gluing removes a duplicate, leaving only a single copy of the data. Fig. 3 shows the gluing for the two documents. In author.xml, for example, a book element is duplicated. We remove one copy by gluing the two subtrees together (shown by dotted lines), and also move the edge from the book element to the remaining copy of the author element (shown by dashed lines).

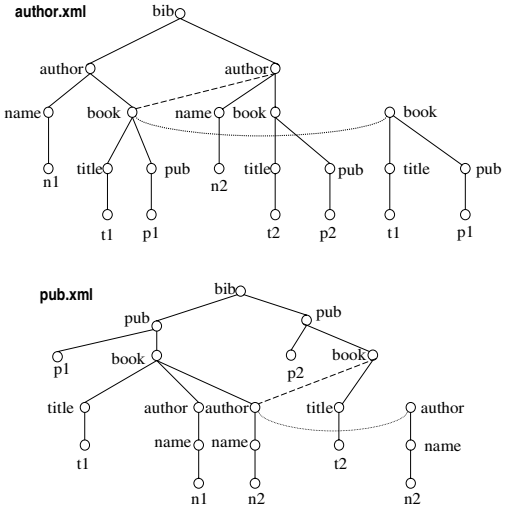


**Fig. 3.** Node gluing

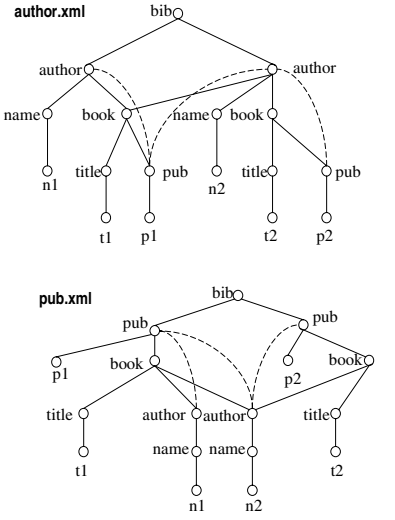


**Fig. 4.** Node connecting

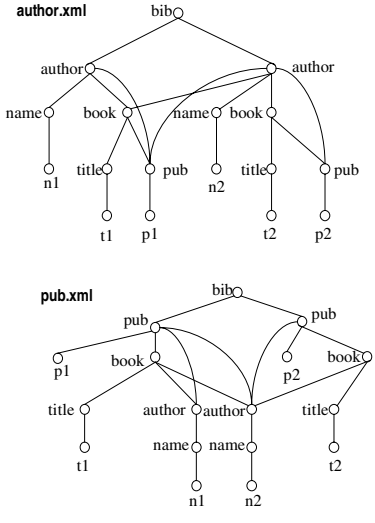


**Fig. 5.** The final graphs

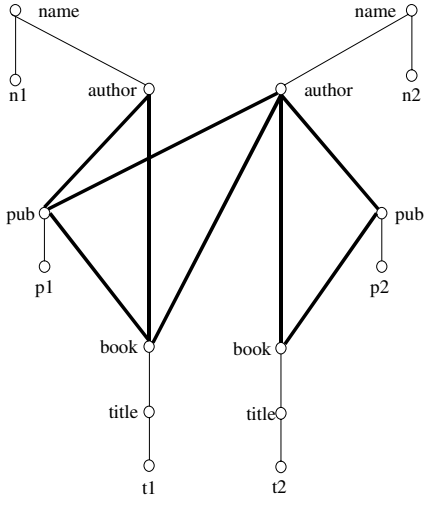


**Fig. 6.** A graph isomorphic to the two final graphs

The next step is to add edges between "related" nodes. In author.xml, authors are related to the books they wrote, and also to the publishers that publish those books. A tree can only (directly) capture relationships between parent and child nodes. The proposed semantics represents every such relationship with an edge, hence creating a graph that will usually contain cycles. We call the process of relating nodes as *node connecting*. Fig. 4 illustrates node connecting (shown by dashed lines). In the figure,

only author, name and publisher nodes are connected. To reduce clutter in the depicted graphs we have chosen not to represent some of the relationships. A connection between the n2 name node and the t2 title node, for example, is not shown. This connection can be inferred because there is a one-to-one correspondence between book and title (as well as author and name). Hence, any node connected to a name node is also connected to the corresponding (its parent) author node. How do we decide which types of nodes to connect and which not to? It depends on the possible parent-child relationships in forests to be semantically compared. For example, since author is always a parent of a name in any possible forest that we compare, we only connect author, but not name, with other types of nodes. On the other hand, since an author node can be either a parent or a child of a book node, we need to make connection between them in the graph. Not adding these edges keeps Fig. 4 less cluttered. More importantly, it saves a certain amount of cost (depending on the property of the data) not to physically materialize these edges. Logically, however, these edges are present in the graph.

Note that the root node bib is, as we would infer, equally related to all nodes in each document. For ease and clarity of presentation, we choose to remove this bib node in both graphs. The final graphs resulting from node connecting are shown in Fig. 5. The two graphs in Fig. 5 are isomorphic. To illustrate this more clearly, a graph that is isomorphic to both is depicted in Fig. 6. It is semantically equivalent to the two initial trees because neither subtree gluing nor node connecting changes the "meaning" of the data. It is also a "minimal" form of the original trees in the sense that duplicate data has been eliminated. The graph in Fig. 6 (as well as the graphs in Fig. 5) is a *canonical* representation of the two initial trees, because it is semantically equivalent to the original data, yet syntactically minimal.

Formally, deriving this semantics consists of the following two steps.

1. **Node Gluing:** Two nodes are glued together if and only if they are identical. i.e., they are of the same type and their identifiers evaluate to the same value. The idea is that, if $u$ and $v$ are identical, then it is only necessary to keep one copy. We can replace every edge $(v,y)$ with $(u,y)$ and then remove $v$. Adding new edges to the forest may result in cycles. Thus gluing produces a graph in general. When nodes are glued in this process, the size of $V$ decreases, and the size of $E$ does not increase (and may decrease). As we can see, semantically comparing two forests is only possible when given the set of identifiers for all types of nodes. Identifiers carry the information about how nodes are related, and are crucial to reason about data semantics.
2. **Node Connecting:** In the next step, related nodes are connected. The idea is that every pair of related nodes is now explicitly identified by an edge that connects them. Before node connecting, a pair of related nodes may or may not be adjacent, while semantically whether they are connected or not should not make any difference. Connecting effectively changes a tree to a graph by adding edges. The number of edges in $E$ may either increase or decrease depending on the specific situation. There is no change to $V$, $\Sigma$, $L$, or $C$.

## 4 Compaction

With a semantics for XML data defined, a formal discussion on compaction is now possible. Compaction aims to transform a forest to a smaller forest. The two forests will have the same semantics, i.e., the forest-to-graph translation maps them to the same graph. Compaction can be achieved by changing the structure of a forest. A *restructuring* is a transformation that changes the structure of the forest but keeps its semantics intact. A forest can be restructured by mapping it to its canonical graph, and then creating a semantically equivalent forest with a different structure. Ideally, the restructuring will yield a forest that is *smaller* in size than the original. Here we employ a structural specification called a *signature* that designates the target's structural characteristics. For example, the signature for pub.xml is

bib#pub#book#(author#name,title)

in which the symbol # denotes parent-child relationship and siblings are separated by commas and enclosed within a pair of parentheses.

We have devised a restructuring algorithm that takes a canonical graph and a target signature as input, and outputs a new forest that conforms to the specification. (Due to space limit, the detail is omitted in this paper; it can be found at [8]). Essentially, this restructuring algorithm is an inverse of the combination of node gluing and node connecting. In changing the canonical graph to a forest, the restructuring algorithm *disconnects* and *unglues* nodes. It disconnects since the output has to contain no cycle, and it unglues (makes duplicates) since the semantics encoded in every edge in the canonical graph must be faithfully preserved.

In restructuring, different target signatures will yield forests of different sizes. To find the most compact forest among them, we could simply enumerate all the possible target forests. However, this is computational intractable. The number of different target signatures is more than exponential.[1]

While in general the problem is hard, there is a simple technique to generate a compact forest for some forests. The idea is to take advantage of the *cardinality ratio*. The ratio characterizes the relationship between pairs of element types as one of the following: one-to-one, one-to-many, or many-to-many. For example, the relationship between publisher and book is one-to-many: a book is published by exactly one publisher but a publisher publishes many books. On the other hand, the relationship between book and author is many-to-many.

Cardinality ratio may come with the data as a predefined constraint; if not, it can be quickly determined by traversing the canonical graph. (In contrast, we do not infer type identifiers and assume they must be given.) Table 1 shows the cardinality ratios for the example graph. The relationship between author and name is one-to-one (recall that authors are glued by name, hence each author is associated with a single name, and vice-versa). Author to book is many-to-many since an author can write many books, and a book can have many authors. (Note that exact, average ratios could be computed, e.g., 4.2 to 2.7.)

---

[1] Given a label set of size $n$, suppose the number of distinct unordered trees is $t(n)$ and the number of distinct unordered forests is $f(n)$, we have,

$t(1) = 1$, $t(n) = (2n\text{-}2) \bullet t(n\text{-}1) = (2n\text{-}2) \bullet (2n\text{-}4) \ldots 2 \bullet t(1) = (2n\text{-}2) \bullet (2n\text{-}4) \ldots \bullet 2$, and

$f(1) = 1$, $f(n) = (2n\text{-}1) \bullet f(n\text{-}1) = (2n\text{-}1) \bullet (2n\text{-}3) \ldots 3 \bullet f(1) = (2n\text{-}1) \bullet (2n\text{-}3) \ldots \bullet 3$.

**Table 1.** Cardinality ratios for the example graph

| | pub | title | book | name |
|---|---|---|---|---|
| **author** | n-m | n-m | n-m | 1-1 |
| **name** | n-m | n-m | n-m | |
| **book** | n-1 | 1-1 | | |
| **title** | n-1 | | | |

The key to achieving compactness is to focus on types that are related in one-to-many relationships. Specifically, assume types *X* and *Y* are in a one-to-many relationship. Then a target signature that has *X* above *Y* leads to a forest that is more compact than a forest with *Y* above *X*. Consider the example of publisher and book, which have a one-to-many relationship. If publishers are above books in the target signature, then in the target forest there are no duplicate publishers (or duplicate books). Every book is placed under the publisher to which it belongs. If books are above publishers in the target signature, then the same publisher may be duplicated several times. Such a forest is less compact and hence need not to be considered.

The technique for generating a target signature for a compact forest begins by considering one-to-one relationships. One side of the relationship is made a child of the other side. If one side is involved in gluing the other, then it should be made the child, otherwise either side can be made the child. Consider book and title in the example graph. Their relationship is one-to-one. Furthermore, book is glued using title. Hence book should be a parent of title in the potentially compact output. The target signature after this step is book#title and author#name. Next, one-to-many relationships are processed by making the one side the parent. In the example, after considering the one-to-many relationships, the target is pub#book#title and author#name. Finally, only many-to-many relationships remain. The remaining types are placed as high as possible in the forest. In the example graph, this means that author is made a child of book resulting in the signature pub#book#(title,author#name).

Once the target signature has been generated, the original forest is restructured using the target signature. The restructured forest may or may not be smaller, i.e., the technique does not generate the *most compact* forest. Finding the signature that leads to the most compact restructuring is theoretically intractable. However, we expect the technique outlined above to lead to a "reasonable" target signature in practice. The technique can be further refined by utilizing the average cardinality ratio of many-to-many relationships, e.g., if the ratio is thirty-to-two, then the two side of the relationship should be made the parent. But such refinements are beyond the scope of this paper.

To gauge how well compaction performs on real-world data, we did an experiment to compact DBLP data. The test data has a size of 309KB and contains 7312 elements. Restructuring the data using a compact signature yields a 252KB data collection that contains 5441 elements. The compacted data has an 18% reduction in file size and a 25% reduction in number of elements. The detail of the experiment can be found at [8].

## 5 Related Work

Compaction for XML is similar to XML compression in the sense that they both aim to describe the same information with shorter representation. However, compaction differs from usual compression since the output has to retain XML syntax. To the best of our knowledge, the problem of compaction has not been previously researched. In this section, we briefly review general and XML-specific compression techniques, and relate them to compaction when pertinent.

Most modern data compression techniques have their genesis in the Huffman algorithm [767] or the LZ77 algorithm [7]. Huffman coding is *statistical*; it assigns shorter codes to more frequent characters and longer codes to less frequent ones. Popular data compression tools such as gzip and pkzip are based on LZ77. A later version of LZ77, the LZW algorithm [6], is more suitable for practical implementation. The essence of the LZ77 family of compression techniques is to store repetitive sequences just once. Any repetition of a sequence that previously occurred is replaced by a pointer to that sequence. Such techniques are called *pattern-based*.

Specialized compressors take advantage of the specific properties of the data to be compressed. For XML data in particular, several compression techniques have been proposed. The earliest such work is XMill [2]. Incorporating existing compressors, XMill compresses XML structures and values separately, uses type specific compressors for different types of data, and allows user-defined compressors for domain specific data-types. Data compressed by XMill cannot be directly queried; doing so would entail a complete decompression. XGrind [4] and XPRESS [3] are both compressors that support direct query evaluations on compressed XML data. XGrind uses a compression scheme based on Huffman coding, while XPRESS adopts an encoding method called reverse arithmetic encoding. It is worthwhile to note that both compression techniques are *homomorphic* because the structure of the original XML data is preserved in the compressed XML data. In contrast, an important compaction technique proposed in this paper, restructuring, changes the structure of the original data. Homomorphism is important for the compressed data to be efficiently queried. Compaction, on the other hand, is useful to ascertain the semantic redundancy in the data. Compacted XML data is not supposed to be queried directly by the query intended for the original data. Among the three compressors, XGrind is the only one that tries to utilize schema information such as a DTD to enhance the compression ratio. In comparison, finding an appropriate schema (target signature in our situation) is the goal of compaction. To enhance compaction ratio, knowledge of identifiers in the original data is required, and knowledge of cardinality ratio is helpful.

## 6 Conclusion

XML compaction aims to produce a smaller, compact XML forest, without losing information. This paper develops a formal framework for the compaction of XML data. It first formalizes XML data by introducing a forest data model and defining types and identifiers. A translative semantics for XML is then presented. This semantics translates an XML data collection to a canonical graph, depending on a given set of identifiers. Data collections that translate to the same canonical graph are deemed

to have the same semantics. Based on this formalization, two compaction techniques, restructuring and grouping, are discussed. Though finding the *most* compact forest is computationally prohibitive in general, we developed simple techniques to find a *more* compact forest at low cost using restructuring or grouping.

In future we plan to explore the relationship between compaction and compression. General compression techniques are not confined to produce the same file format as the input. Hence, it is reasonable to expect that they can achieve a better compression than compaction. However, a file can be first compacted and then compressed. Does combining the compaction with compression produce better performance than compression alone? An interesting work is to examine this problem on both the theoretical and experimental grounds.

## References

1. D. Huffman. A Method for Construction of Minimum-Redundancy Codes, Proc. of IRE, September 1952.
2. H. Liefke and D. Suciu. Xmill: An Efficient Compressor for XML Data, SIGMOD Conference, 2000.
3. J. Min, M. Park, and C. Chung. XPRESS: A Queriable Compression for XML Data, SIGMOD Conference, 2003.
4. P. M. Tolani and J. R. Haritsa. XGRIND: A Query-friendly XML Compressor, ICDE 2002.
5. W3C. Document Object Model (DOM), www.w3.org/DOM.
6. T. Welch. A Technique for High-Performance Data Compression, Computer, pp. 8-18, 1984.
7. J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, 23:3, pp. 337-343, 1977.
8. http://www.eecs.wsu.edu/~cdyreson/pub/compaction.