Updates on Grammar-Compressed XML Data

Alexander Bätz, Stefan Böttcher, and Rita Hartel

University of Paderborn, Computer Science, Fürstenallee 11, 33102 Paderborn, Germany
{laures,stb,rst}@uni-paderborn.de

Abstract. In this paper, we present updates on CluX, a grammar-based XML compression approach based on clustering XML sub-trees. We show that updates on CluX-compressed data can be performed faster than decompressing the data, loading it into main memory and compressing it. Furthermore, we show how to support fast multiple updates, e.g. performing 100 updates in parallel is more than 70 times faster than 100 single updates.

Keywords: updating compressed XML data, grammar-based compression.

1 Introduction

Motivation: XML is widely used in business applications and is the de facto standard for information exchange among different enterprise information systems. Whenever the amount of processed XML data is a bottleneck, it is desirable that applications can directly query and update compressed XML data without having to decompress the data before accessing it.

There have been different contributions to the field of XML compressors generating queryable XML representations, that range from encoding-based [1] to schema-based [2], [3] to DAG-based [4] to grammar-based [5] compressed representations. We follow the grammar-based XML compression techniques, and we discuss how an XML compression technique, called CluX, can be extended by updates. Like the big majority of the XML compression techniques (e.g.[1],[2],[3], [5],[6],[7],[8],[9],[10],[11]), we assume that textual content of text nodes and of attribute nodes is compressed and stored separately and focus here on the compression of the structural part of an XML document.

Contributions: This paper proposes an approach to perform updates on grammar-compressed XML data directly, i.e., without prior decompression of the compressed data. Furthermore, our approach allows to perform several updates in parallel in such a way that e.g. performing 100 updates in parallel is more than 70 times faster than performing 100 updates sequentially.

We have implemented and evaluated updates on the compressed data. Our results show that it is not only possible to perform parallel updates on the CluX compressed data directly, but furthermore that in many cases, these updates can be performed in less time than it would take to decompress the compressed data, load the XML document, and compress the data again.

For simplicity of this presentation, we restrict it to XML documents containing only element nodes, i.e., attributes are regarded as special element types. Note

A.A.A. Fernandes et al. (Eds.): BNCOD 2011, LNCS 7051, pp. 154–166, 2011.

however that our implementation can handle full XML documents including attributes, text values, etc..

Paper Organization: The remainder of this paper is organized as follows. Section 2 describes the basic concept of grammar-based compression, i.e., how an XML tree can be stored in a more space saving way by sharing similar structures, it explains how these shared structures can be represented by patterns being used in tree grammars, and it describes how paths in XML document trees correspond to paths in tree grammars. Section 3 describes how updates can be performed on the compressed data directly and discusses the phases of performing updates: combining the update paths to an update DAG, isolating the update DAG from the grammar, updating isolated nodes, and sharing of identical sub-trees. Section 4 describes the evaluation of the presented update method. Section 5 compares our contribution to related work. Finally, Section 6 summarizes our contribution.

2 Sharing Similar Trees

2.1 The Paper's Example Document

To simplify the following presentation, we do not distinguish between an XML node and its label. The following example is used for explaining the idea of grammar-based compression and to give a visual representation of our idea of direct updates on the compressed data.

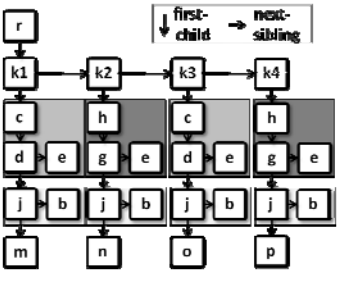

Fig. 1. Document tree of an XML document D with repeated matches of patterns

Fig. 1 shows an example XML document D represented as a binary tree, where e.g. r's first-child is k1 whose next-sibling is k2. This XML document tree can be generated by the following grammar using the non-terminal S as the start symbol and the symbol ε as the empty sub-tree, i.e., the right hand side of the grammar rule is a term representing the pre-order notation of the binary tree given in Fig. 1:

S → r(k1(c(d(j(m(ε,ε),b(ε,ε)),e(ε,ε)),ε), k2(h(g(j(n (ε,ε),b(ε,ε)),e(ε,ε)),ε),
k3(c(d(j(o (ε,ε),b(ε,ε)),e(ε,ε)),ε), k4(h(g(j(p (ε,ε),b(ε,ε)),e(ε,ε)),ε),ε)))),ε)

Grammar 1: Grammar corresponding to the binary tree of Fig. 1

2.2 The Idea Behind Sharing Similar Trees

The simplest grammar-based XML compressors are those compressors that share identical sub-tree structures, such that the compressed grammar represents the minimal DAG of the XML tree [4].

Approaches like binary DAG compression, that share identical sub-trees T in an XML document D replace repeated occurrences of T in D by replacing each occurrence of T in D with a non-terminal N and adding a grammar rule that defines N to be a non-terminal that represents T.

In Grammar 1, there are four matches for each of the two patterns b(ε,ε) and e(ε,ε). Therefore, these matches can be replaced by the non-terminals B and E respectively, such that we get the following grammar:

S → r(k1(c(d(j(m(ε,ε),B),E),ε), k2(h(g(j(n (ε,ε),B),E),ε),
k3(c(d(j(o(ε,ε),B),E),ε), k4(h(g(j(p (ε,ε),B),E),ε),ε)))),ε)
B → b(ε,ε)
E → e(ε,ε)

Grammar 2: Grammar corresponding to the binary DAG of the XML tree of Fig. 1

Our approach goes beyond the idea of DAG compression and uses a parameterized grammar for sharing not only identical sub-trees, but even similar sub-trees. It follows the idea of grammar-based compression as it was introduced in BPLEX [5].

When looking for similar sub-trees having small differences, we find the three different patterns shown in Fig. 2(b) in the document tree D of Fig. 1: one pattern consisting of the nodes c, d, and e, another pattern consisting of the nodes h, g, and e, and a third pattern consisting of the nodes j and b respectively. For each of these patterns, there exist several matches in D which are highlighted in Fig. 2(a). Although the matches of the patterns have identical inner nodes, they cannot be shared in a DAG because the leaf nodes with labels m, n, o, or p respectively differ from each other.

Fig. 2 (b) shows the patterns JB(X), CDE(X) and HGE(X), which consist of the nodes (j and b), (c, d, and e), and (h, g, and e) respectively. The nodes j, d, and g have a parameter 'X' as first-child.

The compression achieved by replacing the repeated patterns with a non-terminal can be seen when comparing Grammar 2 with Grammar 3 shown below. We express the pattern JB(X) of Fig. 2(b) by one grammar rule with the left hand side JB(X), where the parameter 'X' is being used for referencing the different child nodes m, n, o, and p of the j-nodes. This grammar rule is being used, e.g. when the term j(m(ε,ε),B) occurring in the rule S of Grammar 2 is replaced with the term JB(m(ε,ε)) occurring in the rule S of Grammar 3. Here, j(m(ε,ε),B) is called a *match*, and JB(m(ε,ε)) is called a *corresponding instantiation* of the pattern JB(X).

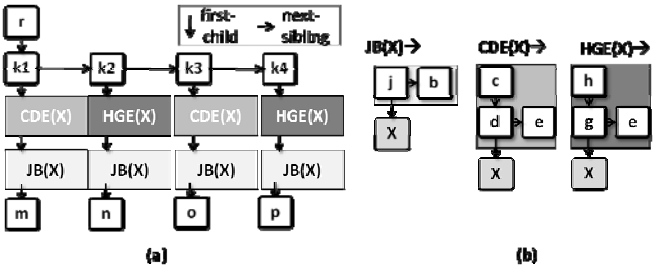

Fig. 2. (a) Example document of Fig. 1 with repeated patterns replaced by non-terminals. (b) Repeated patterns

By replacing each match with a corresponding instantiation, we get the following grammar, Grammar 3, which is more compact than Grammar 2:

S → r(k1(CDE(m(ε,ε)), k2(HGE(n(ε,ε)), k3(CDE(o(ε,ε)), k4(HGE(p(ε,ε)),ε)))),ε)
CDE(X)→ c(d(JB(X),e(ε,ε)),ε)
HGE(X)→ h(g(JB(X),e(ε,ε)),ε)
JB(X) →j(X,b(ε,ε))

Grammar 3: A grammar sharing patterns by using parameterized rules

All terminal nodes except ε have two parameters, i.e. the first-child and the next-sibling. However, non-terminal nodes may have an arbitrary number of parameters.

2.3 Node Selection by Grammar Paths

The grammar path (GP): Each path to a selected node in any given XML document D corresponds to exactly one grammar path (GP) in the compressed grammar G of D. Intuitively, GP describes not only which grammar rules are called to find the selected node, but also from where in a given grammar rule, the next grammar rule is called. For this purpose, GP contains an alternating sequence of grammar rule names and index positions within these grammar rules of the occurrences of non-terminals Ni calling the next grammar rule. Additionally, the last number in GP is the index of the terminal symbol corresponding to the selected node in the last grammar rule collected by GP.

For example, if we apply the query /k2//b to Grammar 3, GP is initially [S], i.e. contains only the start rule. When k2 is found in the start rule S, no other grammar rule has been used, thus GP is still [S]. When the search for a descendant b of k2 continues via k2's first-child, the 2nd non-terminal in the rule for S, i.e. HGE(X), is called, and GP now is [S,2,HGE(X)]. Later, to find the first-child of g within the grammar rule for HGE(X), the 1st non-terminal, i.e. JB(X), is called. Finally, within the grammar rule for JB(X), we pick the 2nd terminal symbol, i.e. the symbol b, to complete GP. This grammar path (GP), i.e. [S,2,HGE(X),1,JB(X) : 2], corresponds to the b-node selected by the query /k2//b.

A formal definition of grammar paths however omitting the rule names is given in [12].

3 Parallel Updates on the Compressed Data

3.1 Basic Update Concepts and Parallel Update Problem Definition

The grammar DAG (GD): The grammar DAG (GD) visualizes from which non-terminal position within which grammar rule other grammar rules are called. In the grammar DAG (GD), there is one node Ni for each grammar rule Gi, and there is one edge (N_1,I,N_2) from node N_1 to node N_2 for each occurrence of a call of the grammar rule G2 within the grammar rule G1, such that (counted from left to right) the I^{th} outgoing edge of N_1 refers to N_2, if G2 is the I^{th} non-terminal occurring in the right-hand side of G1. For example, the GD of Grammar 3 is shown in Figure 3.

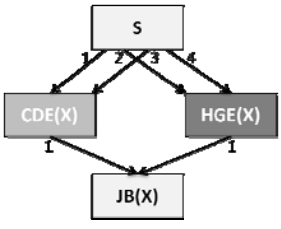

Fig. 3. Grammar DAG (GD) of Grammar 3

Each prefix P of a grammar path GP=[P:t] through the given grammar G corresponds to one path in the GD. Based on this observation, we use GD for parallel updates on G.

For any given current context node ccn of the XML document, we simulate the following elementary update operations on the compressed grammar G, as more complex update operations can be constructed from these elementary update operations: delete ccn (del), re-label ccn with the new label y (reLTo(y)), insert SubTree as first-child of ccn (newFC(SubTree)), insert SubTree as next-sibling of ccn (newNS(SubTree)).

The Update Path (UP): For each elementary update operation U ∈ { reLTo(z), delete, newFC(SubTree), newNS(SubTree)) } that is applied to a single selected XML document node being represented by a grammar path GP=[N_1,I_1,…,N_{k-1},I_{k-1},N_k : t], we define a corresponding update path UP = [(N_1',I_1,N_2'), …, (N_{k-1}',I_{k-1}',N_k') , (N_k',t,U)] . The update path contains a new copy N_i' of each grammar path node Ni. Furthermore, we represent edges in the update path as triples (Start,Index,End). Finally, we add an edge (N_k',t,U) from the copy N_k' of the last non-terminal N_k with the index position t of the terminal symbol in the grammar rule represented by N_k' to which the update operation U shall be applied. As a result, the final node of the update path contains the update operation U applied to the t^{th} terminal of N_k'.

For example, let the update operations be re-labeling all the nodes selected by //h//b to z. Then, we get the update paths [(S',2,HGE(X)'), (HGE(X)',1,JB(X)'), (JB(X)',2,reLTo(z))] and [(S'',4,HGE(X)''), (HGE(X)'',1,JB(X)'') , (JB(X)'',2,reLTo(z))].

The Parallel Update Problem Definition: Given a set of update operations, the parallel update problem is to compute a DAG of update paths and to isolate the DAG of update paths from the grammar G in parallel in order to keep the compressed grammar small and in order to keep the update process fast.

3.2 First Step of the Parallel Update Process: Constructing an Update DAG (UD)

Updating the grammar-compressed data is done by a two-step approach on the given grammar G.

In a first step, we navigate through G and identify the paths that have to be updated and combine them to an update DAG. Instead of collecting all the update paths individually while navigating through G, we combine all the update paths having a common prefix to construct an update tree.

To combine a collection of update paths into an update tree, first, all copies of the start node, i.e. S' and S'' in the previous example, are renamed to a single copy S' of the start node. This reflects the fact that all update paths that are combined into an update tree start at the same root node S'.

Thereafter, each set of common prefixes of multiple update paths is combined to a common prefix in the update tree as follows. Whenever the update tree contains two edges (N_i,I,N_j) and (N_i,I,N_k) where neither N_j nor N_k contains an update operation, the node N_k is renamed to N_j. This reflects the fact that both edges represent the unique I^{th} occurrence of a non-terminal in the grammar rule represented by N_i.

Continuing the previous example, the update tree has a common start node S', but has no common edge of its two update paths.

In addition to combining updates with common prefixes within the update tree, we transform the update tree into an update DAG (UD). The UD is constructed bottom up from the update tree by sharing equal sub-trees. Two leaf nodes within the update tree are equal, if they have the same label, i.e. they contain the same update operation. Two inner nodes U_1 and U_2 of the update DAG are equal if they are copies of the same grammar rule and have similar outgoing edges, where two outgoing edges (U_1, I_1, U_3) and (U_2, I_2, U_4) from nodes U_1 and U_2 respectively are similar if $I_1=I_2$ and $U_3=U_4$.

Continuing the previous example, the sub-DAG rooted in HGE(X)' is equal to the sub-DAG rooted in HGE(X)''. As similar edges are stored only once in the update DAG (UD), UD contains the edges { (S',2,HGE(X)'), (S',4,HGE(X)'), (HGE(X)', 1, JB(X)'), (JB(X)',2,reLTo(z)) } and is shown in Figure 4.

3.3 Second Step of the Parallel Update Process: Isolating UD from GD

The second step of the parallel update process is to isolate the update DAG (UD) from the grammar DAG (GD) by isolating all the update paths contained in UD from GD in parallel. UD isolation is done by combining UD and GD into single DAG, called the extended update DAG (EUD), which is done by adding edges from certain UD nodes to certain GD nodes, such that after the extension, the EUD represents the result of isolating the original UD from GD. The UD isolation and update execution procedure is summarized in Algorithm 1 and is described in the following.

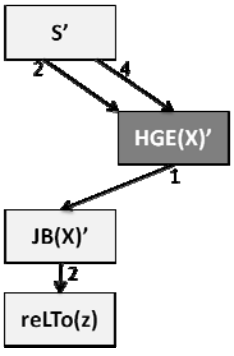

Fig. 4. Update DAG for re-labeling all the nodes of the document shown in Fig.1 that are selected by the query //h//b to z

```
(1)   for each non-leaf node U_i in UD do
(2)   {  N_i = corresponding_GD_node( U_i ) ;
(3)      for each edge ( N_i, I, N_k ) in GD
(4)         if (no edge (U_i, I, U_k) to any node U_k exists in UD) add (U_i, I, N_k) to UD
(5)   }
(6)   EUD = UD ;
(7)   Perform updates on EUD as described in Section 3.4.
(8)   Share identical nodes on EUD as described in Section 3.5.
(9)   Top-down for each node Ni in GD do
(10)     if ( Ni has no incoming edges ) delete Ni and all outgoing edges of Ni
(11)  return EUD ;
```

Algorithm 1. Update DAG (UD) isolation from a grammar DAG (GD)

For each non-leaf node U_i in the update DAG, let N_i be the corresponding node in the grammar DAG, i.e., U_i and N_i belong to the same grammar rule Gi (line (2)). Note that different nodes of the update DAG may correspond to the same node N_i. For each outgoing edge (N_i, I, N_k) of N_i for which no edge (U_i, I, U_k) exists in the update DAG, add an edge (U_i, I, N_k) to the update DAG (lines (3)-(4)). That is, an outgoing edge from U_i to the GD node N_k representing Gi's I^{th} non-terminal is added to each UD node U_i for which an edge to a UD node U_k representing Gi's I^{th} non-terminal does not yet exist, such that finally the UD represents the same number of grammar paths as the GD. On the UD with this extensions, called EUD (line (6)), the update and sharing operations described in the following sections are performed. After computing the EUD and performing all the update and sharing operations, some GD nodes and GD edges are reachable by a path from the UD root and others are not. As the non-reachable GD nodes and GD edges are useless, they are deleted from GD (lines (9)-(10)). The remaining and returned extended update DAG (line (11)) represents the result of isolating the original UD from the GD and performing the updates on the resulting EUD.

Fig. 5 shows the continued example of the isolation process for re-labeling the nodes selected by the XPath expression //h//b to z. The input of the isolation process is the grammar DAG (as shown in Fig. 3) and the update DAG (UD) shown in Fig. 4 and containing the following set of edges: { (S',2,HGE(X)'), (S',4,HGE(X)'), (HGE(X)', 1, JB(X)'), (JB(X)',1,reLTo(z)) }.

For the root node S' of the UD, the corresponding node S of the GD has two additional outgoing edges at the index positions 1 and 3 to the node CDE(X). Therefore, within lines (2)-(4), Algorithm 1 extends the UD with the edges (S',1,CDE(X)) and (S',3,CDE(X)) from S' to CDE(X).

For the UD node HGE(X)', the corresponding node HGE(X) in GD has no additional outgoing edge, i.e. no outgoing edge from HGE(X)' has to be added. That also applies to the UD node JB(X)'.

Now, the isolation phase is completed, and replacing the terminal symbol b with z within the copied grammar rule represented by the node JB (X)' modifies exactly the set of paths selected by the XPath query //h//b to have the new label z, and removes the node reLTo(z) from the EUD.

Finally, the nodes S and HGE(X) and their outgoing edges can be deleted from GD as they are not reachable from any path starting in S'. The resulting UD is shown in Fig. 5(b).

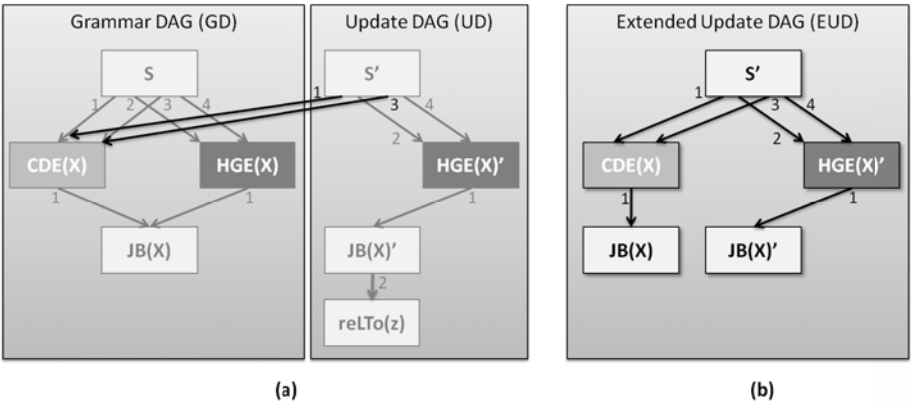

Fig. 5. Isolation of nodes selected by //h//b for re-labeling them to z

Note that we implement UD nodes by copying grammar rules from the grammar represented by GD. For example, Grammar 3 is modified in such a way that the rules S', HGE(X)', and JB(X)' are copied from the rules S, HGE, and JB(X) respectively, and HGE(X)' instead of HGE(X) is called from the S' rule, and JB(X)' instead of JB(X) is called from the HGE(X)' rule. Then, b can be replaced with z within the JB'(X) rule in the following update step.

3.4 Performing the Updates

After UD isolation by extending the UD to a EUD in lines (1)-(6) of Algorithm 1, each edge (U_i, I, U) in EUD to an update leaf node U, i.e. a leaf node of EUD where

U is an update operation, is used for modifying the grammar rule Gi represented by Ui. The update operation U is applied to the I^{th} terminal symbol T of Gi, e.g., for the edge (JB(X)', 2, reLTo(z)), the grammar rule JB(X)' is updated by replacing the 2^{nd} terminal symbol, i.e. b, with z. Depending on the particular update operation U described by the edge (U_i, I, U), we do the following.

If U= reLTo(z)) , we substitute the I^{th} terminal symbol of Gi, i.e. T, with z.

If U=newFC(cst) or U=newNS(cst), i.e., if the update requires inserting a compressed sub-tree cst as a first-child or as a next-sibling of T respectively, we have to set the current first-child or next-sibling of T as the new next-sibling of the root of cst. As we might insert the same sub-tree cst at several places within the original XML document, the most efficient way to do this is to set the next-sibling of the root rule of cst (which is a null pointer, i.e. ε, before the insertion) to a parameter, and to replace the first-child fc or the next-sibling ns of T by a call of the root rule of cst with the parameter value fc or ns respectively.

If we consider for example the node 'k1' as selected node in Grammar 3 and want to insert a node z(ε,ε) as the first-child of /k1, i.e., cst consists of the single rule Z(X) → z(ε,X), we would insert a new rule Z(X) → z(ε,X) into Grammar 3 and replace the current first-child cfc of k1 in Grammar 3 with a call Z(cfc) of this new rule. Thereby, cfc becomes the next-sibling of z. This means, that the call 'k1(CDE(m(ε,ε)),...)' within the start rule S of Grammar 3 is replaced with the call 'k1(**Z**(CDE(m(ε,ε))),...)'.

If U=del, i.e., the update requires deleting the sub-tree having its root in T, we can simply replace T(FC,NS) with T's own next-sibling NS in the grammar rule represented by U_i. If we remove a formal parameter from a rule during the deletion, we have to delete the corresponding actual parameter within each call of the modified rule as well. In the worst case, i.e., if the actual parameter of a rule is defined in the rule represented by S', we have to modify all the rules represented by nodes of EUD that lay on paths from S' to U_i within the EUD.

For example, if we delete the nodes of the sub-trees, the root of which is selected by //d//j, i.e. is the j node with first child m or the j node with fist child o, UD contains the edges (S',1,CDE(X)') , (S',3,CDE(X)') , (CDE(X)',1,JB(X)') , and (JB(X)',1,del). By applying the delete operation to the first non-terminal of the JB(X)' rule, i.e. to j, the right-hand side of this rule is replaced with b(ε,ε). As now the JB(X)' rule does not contain a parameter anymore, the parameter has to be removed from the CDE(X)' rule calling it too. And finally, we have to delete the parameters used in the rule calls of rule CDE(X)' within the S' rule too. By doing this, the first-child nodes of nodes //d//j with labels m and o are deleted as well.

3.5 Sharing Identical Nodes

Although the initial grammar to be updated does not contain anymore sub-structures that can be shared, during the update process new sub-structures are generated that might be similar or identical to already existing sub-structures.

For example, if we re-label in our example the node /k2//g to 'c' and the node /k2//h to 'd', after the UD isolation and the update process, the grammar would contain a rule

$$HGE'(X) \rightarrow c(d(JB(X),e(\varepsilon,\varepsilon)),\varepsilon)$$

which is identical to the rule CDE(X). Therefore, we can delete the rule HGE'(X) and replace each call of it by a call of the rule CDE(X). The grammar using rule HGE'(X) is correct and can be decompressed and processed correctly, but after replacing HGE'(X) with CDE(X), the grammar is more compact, i.e., the compression ratio is optimized. For this purpose, we perform a sharing phase after the updating phase.

In order to find redundant rules, we could compare the modified rule with all existing and modified rules. But this comparison becomes more efficient, when we use the information given by the EUD. Two rules can only be identical, if they call the same sequence of other grammar rules. For the EUD, this means that we only have to compare these rules that have the same sequence of children within the EUD. This reduces the number of comparisons within the sharing phase.

4 Evaluation

All tests were performed on an Intel Core2 Duo CPU P870 @ 2,53 GHz with 4 GB of RAM running our prototype on Java 1.6.

In a first series of measurements, we compared the compression strength of CluX with two other approaches, gzip and bzip2, based on the following XML datasets:

1998statistics (1998 – 656 kB) – Baseball statistics of the year 1998, catalog-01 (C1 – 10.4 MB) and dictionary-01 (D1 – 10.4 MB) – documents generated by the XBench benchmark, hamlet (H – 273 kB) – the Shakespeare play, JST_snp.chr (JST-35.5 MB) – data on the tumor suppressor gene JST, and NCBI_gene.chr (NCBI – 23.0 MB) – data from the National Center for Biotechnical Information, Treebank (TB – 51.9 MB) –a parsed text corpus, and XMark (XM – 111.1 MB) – a document that models auctions.

Usually, CluX compresses best (c.f. Fig. 6), followed by bzip2, and finally followed by gzip.

In a second series of measurements, we have compared the time for direct updates on the compressed data to the sum of the times needed for decompression, loading the uncompressed document as a DOM tree into main memory (i.e., no updates were performed) and recompression when using CluX, bzip2 or gzip as compression tool.

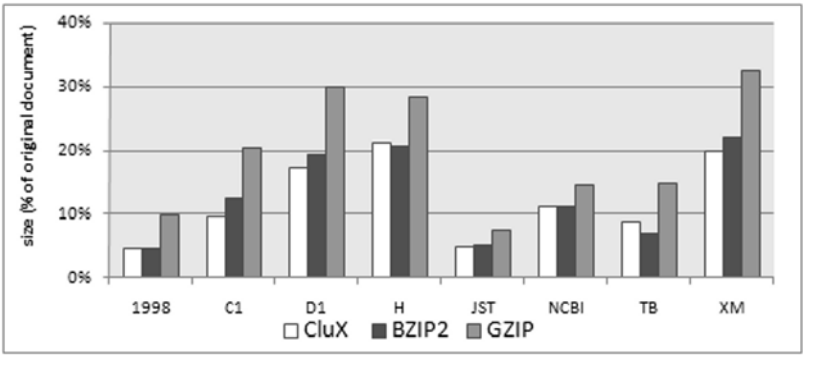

Fig. 6. Compression ratios of CluX compared with bzip2 and gzip

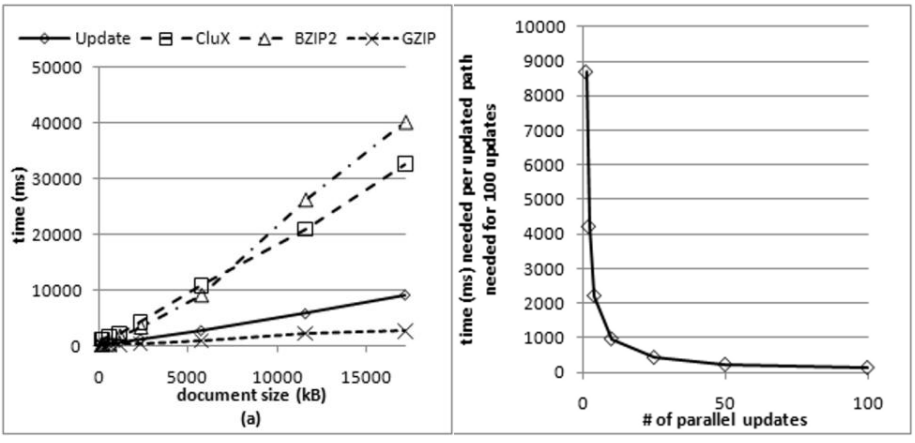

Fig. 7. (a) Update time of CluX compared to compression and decompression times of CluX, bzip2 and gzip, (b) Update time required for a scaling number of parallel updates

With scaling the document size (c.f. Fig. 7(a)), the direct updates on CluX can be performed faster than the compression and decompression of CluX and bzip2. For a document with a size of 15 MB, the update on the compressed data is 3.5 times faster than the decompression and recompression by CluX and 4.4 times faster than the decompression and recompression by bzip2. Only gzip, that reaches a far weaker compression ratio than CluX can be decompressed and recompressed in less time than the update process directly on the compressed data requires. Finally, we have examined the impact of parallel updates compared to sequential updates. For this purpose, we randomly selected 100 paths of the grammar DAG and relabeled the XML node defined by these paths. Fig. 7(b) shows that performing 100 updates in parallel as a multi-update operation is more than 70 times faster than performing 100 updates sequentially.

5 Related Work

Besides generic compressors like gzip, bzip2 or 7zip (based on LZMA) all of which do not allow direct query evaluation on the compressed data, there are several approaches to XML structure compression. XML structure compression can be mainly divided into three categories: encoding-based compressors, schema-based compressors and grammar-based compressors.

The encoding-based compressors allow for a faster compression speed than the other ones, as only local data has to be considered in the compression as opposed to considering different sub-trees as in grammar-based compressors. Examples for encoding-based approaches are the approaches [13], [6], and [7], XMill [8], XPRESS [9], XGrind [14], and [1]. Whereas XMill is not queryable, i.e., it does not support the navigation or the evaluation of XPath queries on the compressed document directly, i.e., without prior decompression, all other approaches are queryable.

Schema-based compression comprises such approaches as XCQ [2], XAUST [15], Xenia [3], and XSDS [10]. They subtract the given schema information from the structural information. Instead of a complete XML structure stream or tree, they only

generate and output information not already contained in the schema information (e.g., the chosen alternative for a choice-operator or the number of repetitions for a *-operator within the DTD). These approaches are queryable and applicable to XML streams, but they can only be used if schema information is available.

XQzip [11] and the approaches presented in [16] and [4] belong to grammar-based compression. They compress the data structure of an XML document by combining identical sub-trees.

An extension of [4] and [11] is the BPLEX algorithm [5]. This approach not only combines identical sub-trees, but recognizes similar patterns within the XML tree, and therefore allows a higher degree of compression. The approach presented in this paper, which is an extension of [17], follows the same idea. But instead of combining similar structures bottom-up, our approach searches within a given window the most promising pair to be combined while following one of three possible clustering strategies. Furthermore, in contrast to [12] and [18], that performs updates by path isolation only sequentially, our approach allows performing updates in parallel which takes only a fraction of time.

6 Summary and Conclusions

We have shown how updates can be performed directly on CluX, a clustering-based compression approach for XML trees, i.e., without the need to decompress the compressed data in advance. As an XML file compressor, CluX compresses on average 70% better than the generic compressor gzip and 5% better than the generic compressor bzip2. CluX compression can be applied to infinite data streams – and in contrast to gzip or bzip2, path queries and updates can be evaluated directly on the compressed representation, i.e., without prior decompression. Beyond other clustering or multiplexing based approaches like e.g. the BPLEX algorithm [12], [5], CluX offers an update DAG isolation technique that allows to perform several updates in parallel, and our evaluation has shown that performing 100 updates in parallel takes significantly less time than performing 100 updates sequentially. Furthermore, our evaluation on a file with a size of 15 MB has shown that performing the updates directly on the compressed data with our update algorithm is more than 3 times faster than decompressing the data first and recompressing it with CluX, and it is more than 4 times faster than the decompression and recompression with bzip2.

We furthermore believe that this technique of performing several updates in parallel on the compressed data directly is not restricted to CluX, but can be extended to DAG-based compressors like [5] and to other grammar-based compressors like e.g. BPLEX [5], the main idea of which is to share similar sub-trees.

References

1. Zhang, N., Kacholia, V., Özsu, M.: A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In: Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, Boston, MA, USA, pp. 54–65 (2004)
2. Ng, W., Lam, W., Wood, P., Levene, M.: XCQ: A queriable XML compression system. Knowl. Inf. Syst., 421–452 (2006)

3. Werner, C., Buschmann, C., Brandt, Y., Fischer, S.: Compressing SOAP Messages by using Pushdown Automata. In: 2006 IEEE International Conference on Web Services (ICWS 2006), Chicago, Illinois, USA, pp.19–28 (2006)
4. Buneman, P., Grohe, M., Koch, C.: Path Queries on Compressed XML. In: Proceedings of 29th International Conference on Very Large Data Bases, Berlin, Germany, pp. 141–152 (2003)
5. Busatto, G., Lohrey, M., Maneth, S.: Efficient Memory Representation of XML Documents. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 199–216. Springer, Heidelberg (2005)
6. Cheney, J.: Compressing XML with Multiplexed Hierarchical PPM Models. In: Proceedings of the IEEE Data Compression Conference (DCC 2001), Snowbird, Utah, USA, p. 163 (2001)
7. Girardot, M., Sundaresan, N.: Millau: an encoding format for efficient representation and exchange of XML over the Web. Computer Networks 33, 747–765 (2000)
8. Liefke, H., Suciu, D.: XMILL: An Efficient Compressor for XML Data. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA, pp. 153–164 (2000)
9. Min, J.-K., Park, M.-J., Chung, C.-W.: XPRESS: A Queriable Compression for XML Data. In: Halevy, A., Ives, Z., Doan, A. (eds.) Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, pp. 122–133 (2003)
10. Böttcher, S., Hartel, R., Messinger, C.: XML Stream Data Reduction by Shared KST Signatures. In: 42st Hawaii International International Conference on Systems Science (HICSS-42 2009), Proceedings (CD-ROM and online), Waikoloa, Big Island, HI, USA, pp. 1–10 (2009)
11. Cheng, J., Ng, W.: XQzip: Querying Compressed XML Using Structural Indexing. In: Hwang, J., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 219–236. Springer, Heidelberg (2004)
12. Fisher, D., Maneth, S.: Structural Selectivity Estimation for XML Documents. In: Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, Istanbul, Turkey, pp. 626–635 (2007)
13. Bayardo Jr., R., Gruhl, D., Josifovski, V., Myllymaki, J.: An evaluation of binary XML encoding optimizations for fast stream based xml processing. In: Feldman, S., Uretsky, M., Najork, M., Wills, C. (eds.) Proceedings of the 13th International Conference on World Wide Web, New York, NY, USA, pp. 345–354 (2004)
14. Tolani, P., Haritsa, J.: XGRIND: A Query-Friendly XML Compressor. In: Proceedings of the 18th International Conference on Data, ICDE, San Jose, CA, pp. 225–234 (2002)
15. Subramanian, H., Shankar, P.: Compressing XML Documents Using Recursive Finite State Automata. In: Farré, J., Litovsky, I., Schmitz, S. (eds.) CIAA 2005. LNCS, vol. 3845, pp. 282–293. Springer, Heidelberg (2006)
16. Adiego, J., Navarro, G., Fuente, P.: Lempel-Ziv Compression of Structured Text. In: Data Compression Conference, Snowbird, UT, USA, pp. 112–121 (2004)
17. Böttcher, S., Hartel, R., Krislin, C.: CluX - Clustering XML Sub-trees. In: ICEIS 2010 - Proceedings of the 12th International Conference on Enterprise Information Systems, Funchal, Madeira, Portugal, pp. 142–150 (2010)
18. Damien, F., Maneth, S.: Selectivity Estimation. Patent WO 2007/134407 A1 (May 2007)