SXSAQCT and XSAQCT: XML Queryable Compressors

Tomasz Müldner[1], Christopher Fry[1], Jan Krzysztof Miziołek[2], Scott Durno[1]

[1] Jodrey School of Computer Science, Acadia University, {tomasz.muldner, 062181f, 900390d}@acadiau.ca
[2] IBI AL, University of Warsaw, jkm@ibi.uw.edu.pl

Abstract: Recently, there has been a growing interest in queryable XML compressors, which can be used to query compressed data with minimal decompression, or even without any decompression. At the same time, there are very few such projects, which have been made available for testing and comparisons. In this paper, we report our current work on two novel queryable XML compressors; a schema-based compressor, SXSAQCT, and a schema-free compressor, XSAQCT. While the work on both compressors is in its early stage, our experiments (reported here) show that our approach may be successfully competing with other known queryable compressors.

1. Introduction

In this paper, we describe two novel techniques to deal with queryable XML compression. Both techniques use the approach borrowed from other XML compressors, which separate the document structure from the text values and attribute values (collectively called *data values*). We then encode the document structure to succinctly store information about the input document, and finally apply appropriate well-known *back-end* data compressors to the document structure and to the *containers* storing the data values. It is well known that on average the structure represents between 10 and 20 percent of the size of the entire document, and the remaining 80 percent represents text and attribute values. However, the main focus of our work is on queryable compression, and so we encode the document structure to support *lazy decompression*, i.e. during the querying process of the compressed document; we attempt to decompress "as little as possible". Well-known XML compressors differ in using various granularity of the container information; some compressors use a single container, while others tend to create many separate containers for related values. The former approach is based on the promise that standard data compressors achieve better results when they get large data sets, but require complete decompression in order to perform a query. On the other hand, the latter approach may suffer from poor compression ratios, but it requires the decompression of only a few (possibly just one) containers. In our approach, we attempt to strike a balance between these two extremes; using containers that will be large enough so that they can be effectively compressed, but at the same time the container structure does not require a full decompression to answer a query. In addition, our design allows both; lazy decompression and

Dagstuhl Seminar Proceedings 08261
Structure-Based Compression of Complex Massive Data
http://drops.dagstuhl.de/opus/volltexte/2008/1673

performing operations directly on compressed data, without any decompression.

Because this is a report on currently on-going work, we do not provide a description of related work and include only a few references to other papers on queryable XML compression. This paper is organized as follows. Section 2 introduces basic terminology, and Section 3 introduces the two XML compression techniques described in this paper. Section 4 provides a detailed description of a grammar-based compression, while a grammar-free compression is described in details in Section 5. Section 6 is on the format of the compressed documents, Section 7 on querying, and Section 8 gives results of testing of our compressors. Conclusions and future work are described in Section 9.

2. Basic Terminology and Related Work

2.1 Terminology

In our paper, by the *document order* we mean the order defined by the occurrence of nodes in the document, or in other words the order in which a SAX parser encounters these nodes (which is the same as in in-order traversal of the document tree). More formally, for two nodes x and y, $x < y$ iff the opening tag of x precedes the opening tag of y. Two absolute paths are called *similar* if they are identical, possibly with the exception of the last component, which is the data value.

When comparing various XML compressors, it is important to provide their various characteristics. In general, compressors can be characterized as:
- *archivers*, which concentrate on compression ratio and speed, while they tend to disregard decompression speed (such compressors are also called *asymmetric*)
- *lossless* XML compressors, for which the only differences between the recreated document and the input document are those permitted by the XML *canonicalization* process (such as the order of occurrence of attributes), and *lossy* XML compressors, for which the only differences between the recreated document and the input document are those permitted by the XML canonicalization process, and, in addition removal *spurious whitespace*
- *online* compressors used in network data exchanges, which concentrate on compression/decompression speed, and online operations (i.e. the decompression may start as soon as the first chunk of the compressed stream becomes available, rather then decompressing offline when the entire decompression stream is available)
- *database applications*, that concentrate on queryable compression with random access, and decompression speed
- *queryable* compressors, that are either *batch compressors*, which know a priori the so-called *workloads* (containing queries that can be used), to build data containers in order to optimize queries, or *interactive compressors*, that can expect any kind of queries.

In addition, compressors may or may not use *indexing* and/or *caching* (to our knowledge, currently there are no such compressor available).

Using the above characterizations, our compressors are interactive,

queryable database applications, which use indexing and caching. In addition, we support both, lossless and lossy compression.

When dealing with any kind of data compression, one compares their compressor with other compressors, using a specific set of input documents. In this paper, we follow [Skibinski] and for our experiments, we use the Wratislavia XML corpus from this paper.

3. Introduction to SXSAQCT and XSAQCT

In this section we provide a general introduction to our compression techniques, and in the following sections we provide more details about each of them. Our compressors are used as follows. First, the input file is compressed, using either lossless or lossy compression. Then the user can start a session, during which the compressed file is queried and/or decompressed to recreate the original file.

3.1. Compression using Schema-based Queryable Compressor: SXSAQCT

The first technique is *grammar-based*, i.e. it assumes that the compressor and the decompressor (as well as the query compressor) *share* the description of the grammar. (However, we also consider a scenario, in which the schema needs to be sent to the decompressor). Since, we use XML Schema to specify a grammar, we call this compressor; Schema-based Queryable Compressor, or SXSAQCT (pronounced *sexact*). Given an acyclic/non-recursive schema S (our future work will concentrate on removing this limitation, and allowing recursive schema) we create a schema tree (T_S). Then, given a document D valid in S, we encode the information about the XML document by performing a single SAX traversal of D and annotating T_S, thereby creating an annotated tree ($T_{A,D}$). At the same time, data values are written to various data containers. Note that $T_{A,D}$ provides a faithful but succinct representation of the input document D (in general, the document tree is short but wide, while $T_{A,D}$ is much narrower). Next, $T_{A,D}$ is compressed, by first writing its annotations to one container, and the tree T_D without annotations (which we call a *skeleton* tree) to another container, and then compressing both streams using standard (possibly different) data back-end compressors, to create the compressor's output C_D.

This approach resembles a *permutation-based* approach, in which a document is re-arranged to localize repetitions. However, in our work, $T_{A,D}$ preserves all information about the ordering of elements, and a single container stores only related data values (specifically, we use a single container to store text values for all *similar* paths. Each container may be compressed using different back-end compressor (depending on the type of value in the container). In other words, our approach is in a sense a *homomorphic approach*, but similar paths are "glued together".

Our schema-based approach is summarized in Figure 3.1. For any schema definition that gives an element s, a *choice*, such as minOccurs != maxOccurs, the element s will have an appended *. Element s is called *dirty* if it has an appended *; otherwise it is called *clean*.

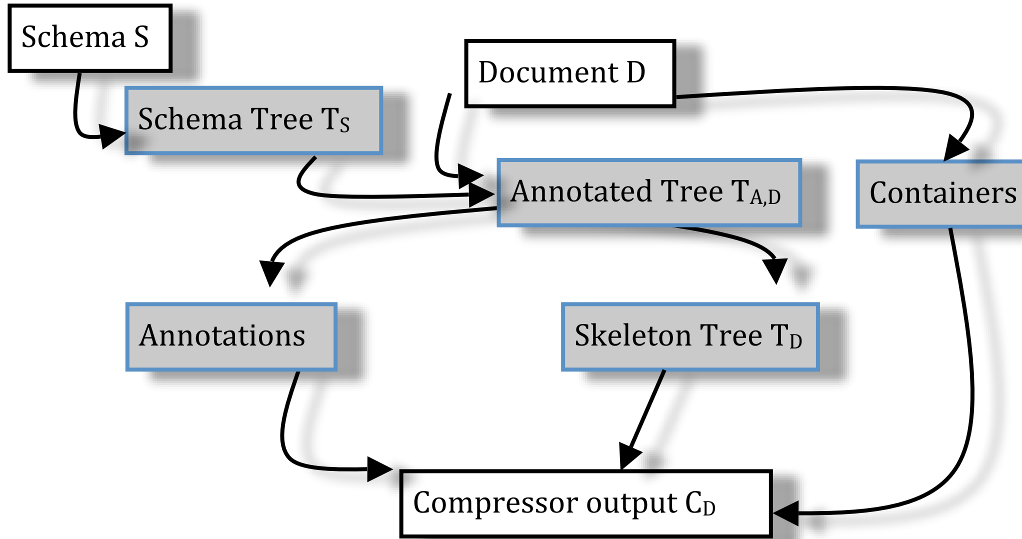

Figure 3.1. The basic architecture of SXSAQCT

Example 1. Consider a schema S, in which the root is labeled 'a'; it's an unbounded sequence of 'b's , where each b is a sequence of an unbounded sequence of 'e's, one 'c', and an unbounded sequence of 'd's. The schema tree T_S is shown in Figure 3.2.

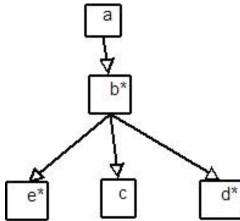

Figure 3.2. The schema tree for schema S

Now, consider the document D valid in S shown in Figure 3.3.

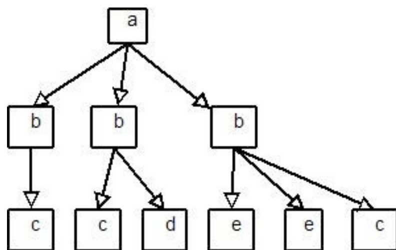

Figure 3.3. The document D valid in S

The annotated tree $T_{A,D}$, which represents D is shown in Figure 3.4.

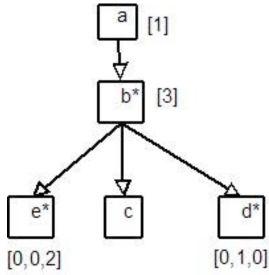

Figure 3.4. The annotated tree $T_{A,D}$

□

Note that our tests have showed that for six files from the Wratislavia XML corpus, the uncompressed tree S_T is very small (ranging from 37 to 313 nodes) and takes little space, specifically it takes between 383 and 3495 bytes. More details on the results of our experiments are provided in Section 8).

3.2. Compression using Schema-free Queryable Compressor: XSAQCT

The second technique is *grammar-free*, and we call our compressor XSAQCT (pronounced *exact*). Here, given an XML document D, we perform a single SAX traversal of D, and create an annotated tree $T_{A,D}$. Therefore, even though in this approach the grammar for D is not provided, we build a concise representation of the structure and the related containers, and then use the same technique as for SXSAQCT. Note that XSAQCT can analyze containers to determine the type of the values stored in each container, in order to decide on a specific back-end compressor to compress this container. The schema-free approach is summarized in Figure 3.5.

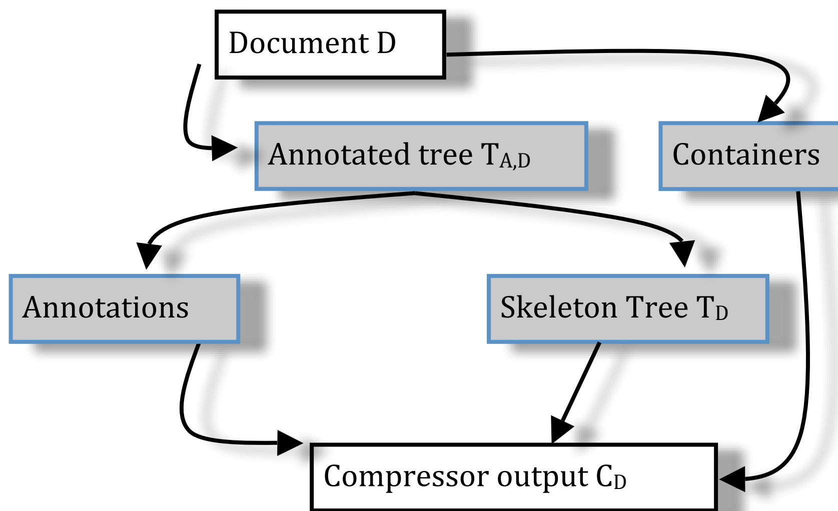

Figure 3.5. The basic architecture of XSAQCT

3.3. Decompression

Both techniques use the same, or very similar, decompression steps. First, the compressed document C_D is decompressed into three kinds of containers; respectively the annotation container, the skeleton tree container, and the containers with text and attribute values. Then, these containers are used to re-create the original document D.

3.4. Attributes and Mixed Content

In this section, we describe handling of attributes and mixed content. Attributes are treated as if they were elements, i.e. their names (preceded by "@"), and annotations are recorded in $T_{A,D}$. Figure 3.6 shows a simple document with various text nodes, including mixed contents (the top of the figure), and the tree $T_{A,D}$ and text containers (the bottom of the figure).

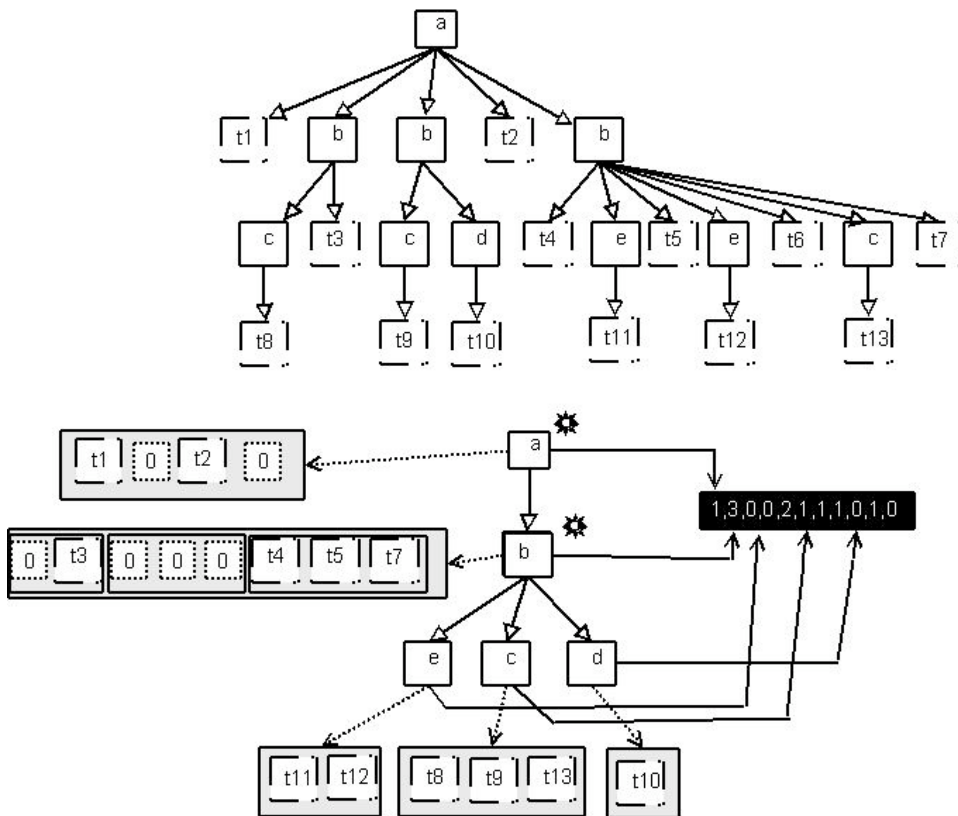

Figure 3.6. Handling mixed content

Nodes of the tree are marked (with an asterisk, in Figure 3.6) if the corresponding element has mixed content, and in such a case empty text (shown as a box containing "0") is inserted in the text container when needed. Note that the figure

3.6 shows the *logical format* for the annotated tree $T_{A,D}$, the actual implementation details are provided in Section 6.

4. SXSAQCT: Schema-based XML Queryable Compressor

In this section, we first describe the algorithm, that for the given *non-recursive* schema S creates a schema tree T_S. Then, we describe the parts of the compressor and the decompressor, which differ from the general description given in Section 3.

4.1. Building a Schema Tree

We start with an example of a schema based on shakespeare.xsd from [Skibinski].

Example 4.1.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="PLAYS">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="PLAY"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="PLAY">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="TITLE"/>
        <xs:element name="FM">
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="FM">
   <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="P"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="P" type="xs:string"/>
  <xs:element name="TITLE" type="xs:string"/>
</xs:schema>
```

The resulting schema tree is shown in Figure 4.1.

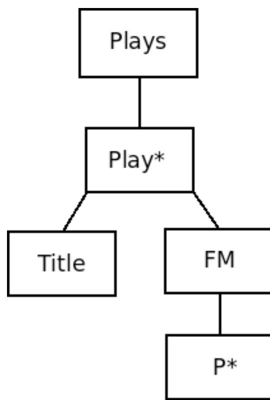

Figure 4.1. Handling mixed content

Algorithm 4.1. Converting XML Schema to a Schema Tree

The schema has a root <xs:schema> then a series of <xs:element>'s. The <xs:element>'s can contains other elements (as well as complexTypes, sequences, etc.) It can also contain references to other elements. If a reference is found, a place holder SchemaNode is created with the type reference, which will later be replaced with a real SchemaNode. We assume that a schema is stored in a file called schemafile.

The algorithm performs the following three steps:

1. Create a graph for every child c of the root <xs:schema>, this will include c's child, grandchildren, etc.
2. Replace all references with "real" nodes
3. One of the children of the <xs:schema> will be the root node of the graph. Find it.

Step 1.

Global variables:

```
SchemaGraph schemagraph;

void parse() {
        Map map = new Map();
        for every child c of <xs:schema> {
                schemagraph = new SchemaGraph();
                generateGraph(child, null);
                //generate the graph of c and store it in global schemagraph
                map.put(child.getName(), schemagraph);
                //store the graph of c in map
        }
}
```

```
void generateGraph(Element element, SchemaNode parentNode) {
        SchemaNode localNode; //the new node to create
        if(element is a <xs:element>) {
          if(element has attribute "name")
                localNode = schemagraph.addNode(element.getName(), parentNode);
          else if(element has attribute "ref")
                localNode = schemagraph.addNode(
                    element.getReference(), parentNode).setType(Reference);
                if(element maxOccurs == "unbounded" || element minOccurs == "0"  ||
                        element minOccurs != maxOccurs) localNode.setDirty(true);
        }
        for every child c of element {
                if(c is a <xs:element>  || <xs:complextype> || <xs:sequence> ||
                        <xs:simpleContent> || <xs:extesion> || <xs:all> || <xs:choice> ||
                        <xs:attribute>) {
                    //so far we ignore everything that is not one of these
    //if localNode was set call generateGraph with localNode as the parentNode of c
                        if(localNode != null) generateGraph(c, localNode);
                        //else use the old parentNode,
                        // this lets us skip over schema elements that don't
                        //get added to the graph like complextype
                        else generate(c, parentNode);
                }
        }
}
```

generateGraph() using the above example is used as follows:

```
generateGraph(<xs:element name ="PLAYS">, null) {
        localNode = schemagraph.addNode(PLAYS, null);
          //so PLAYS is the root of schemagraph
        for every child c of PLAYS {
                c =  <xs:complexType>
                generateGraph(c, PLAYS);
        }
}
generateGraph(<xs:complexType>, PLAYS) {
        for every child c of <xs:complexType> {
                c = <xs:sequence>
                generateGraph(c, PLAYS);
        }
}
generateGraph(<xs:sequence>, PLAYS) {
        for every child c of <xs:sequence> {
                c = <xs:element maxOccurs="unbounded" ref="PLAY"/>
```

```
                generateGraph(c, PLAYS);
        }
}
generateGraph(<xs:sequence>, PLAYS) {
        localNode = schemagraph.addNode(PLAY, PLAYS);
        //play is added as a child of PLAYS
        localNode.setType(Reference);
        localNode.setDirty(true);
}
```

After finishing step 1, the map is shown in Figure 4.2 (where & indicates a reference).

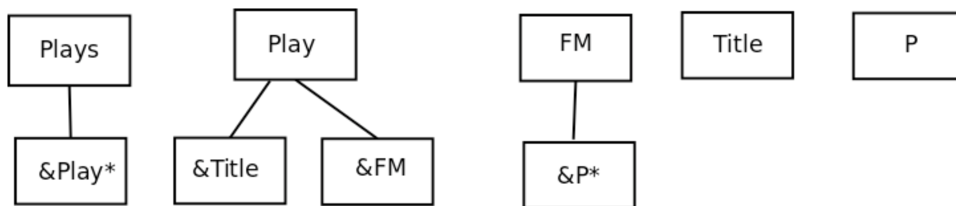

Figure 4.2. Completion of Step 1.

Step 2.

```
//Replace all references with actual nodes
combineGraph(Map map) {
        Queue queue = new Queue;
        for each SchemaGraph g in map
                queue.enqueue(g);
        while(!queue.isEmpty()) {
                SchemaGraph currentGraph = queue.dequeue();
                SchemaNode node = null;
                Iterator<SchemaNode> iter = currentGraph.getDFSIterator();
                while(iter.hasNext()) {
                        SchemaNode temp = iter.next();
                        if(temp.isReference()) {
                                node = temp;
                                break;
                        }
                }
                if(node != null) {
                        queue.enqueue(currentGraph);
                        //the graph may still contain references so try it again
                        currentGraph.replace(temp, map.get(temp);
                        //replace the reference temp with the actual temp from the map
                }
        }
```

}

Table 4.1 shows graphs used in this example.

Table 4.1 Graphs used for shakespeare.xsd.

Queue	**currentGraph**	**temp**	**action**
{Plays, Play, FM, Title, P}	null		
{Play, FM, Title, P}	Plays	&Play	Replace &Play with Play
Plays Play &Title &FM Play &Title &FM FM &P* Title P			
{FM, Title, P, Plays}	Play	&Title	Replace &Title with Title
Plays Play &Title &FM Play Title &FM FM &P* Title P			
{Title, P, Plays, Play}	FM	&P	Replace &P with P
Plays Play &Title &FM Play Title &FM FM P* Title P			

{P, Plays, Play, FM}	Title	null	No action
{Plays, Play, FM}	P	null	No action
{Play, FM}	Plays	&Title	Replace &Title with Title
{FM, Plays}	Play	&FM	Replace &FM with FM
{Plays, Play}	FM	null	Do nothing
{Play}	Plays	null	Do nothing
{ }	Play	null	Do nothing

The resulting graph is shown in Figure 4.3.

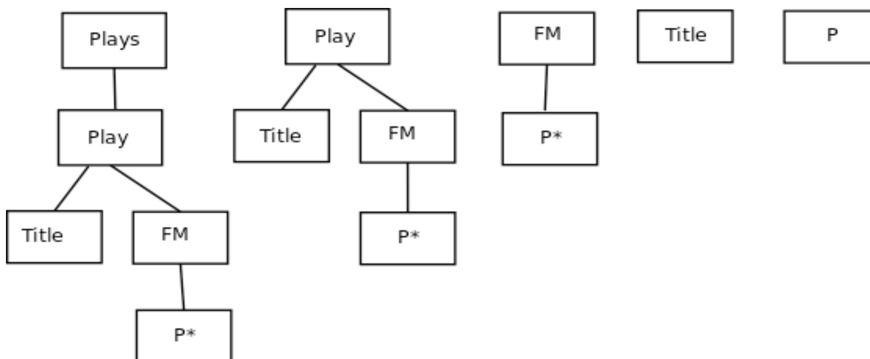

Figure 4.3. Resulting graph for the example.

Step 3
We look at each graph in map and take the largest one (i.e. the graph that contains all the other graphs).

```
SchemaGraph largestGraph(map) {
        for every SchemaGraph g in map
                get the size of g (number of nodes or depth)
        return g with largest size
}
```

largestGraph() for our example return Plays, and is shown in Figure 4.4.

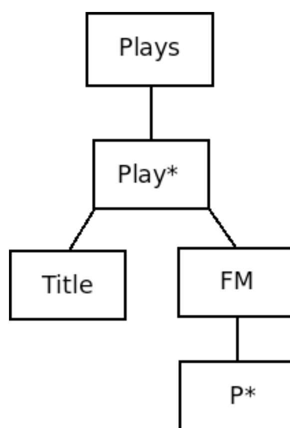

Figure 4.4. Completion of Step 3.
□

4.2. Compression and Decompression

Here, we describe compression and decompression processes for SXSAQCT, see Section 6 for more details of the compression stream C_D. There are two compression modes, respectively resulting in:

1. lower compression rate, but faster query/decompression times. In this mode, the complete annotation tree is stored in C_D, and so the decompressor does not store any information about the schema.
2. higher compression rate, but slower query/decompression times. In this mode, only the partial annotation tree is stored in C_D and so the decompressor needs some information about the schema, in order to restore the complete annotation tree. Here, either a *schema tree* is shared, or only a *schema file* is shared (and so the schema tree needs to be restored). This mode is particularly useful for an environment where multiple documents are submitted for querying and compression, and it is assumed that all these documents are valid in the same schema (the complete annotation tree can be restored when the first document is submitted and then cached).

In the former mode, there are two possibilities:

1. The compressor and the decompressor share the schema file. In this case, the decompressor needs to restore the schema tree
2. The compressor and the decompressor do not share the schema file, but they share the schema tree. In this case, the decompressor needs to restore the schema tree.

Recall, that the annotator SAX-es D and annotates T_S, to create $T_{A,D}$. We start by describing some properties of $T_{A,D}$. Let N be a node of $T_{A,D}$, annotated by $[a_1,\ldots,a_k]$; i.e. ann(N) = $[a_1,\ldots,a_k]$.
Then, the following properties hold:
1) H(D)= H($T_{A,D}$), where H(T)– the height of T
2) For each node N of $T_{A,D}$, with children $c_1,\ldots,c_m$

   A) For i=1,…,m, $|ann(c_i)| = \{N\}$, where {N} = sum of $a_1,\ldots,a_k$
   B) in D, there are {N} nodes, $n_1,\ldots,n_{\{N\}}$ labeled by N, for each such node
      $|ann(n_i)| = \{N\}$
   C) Let $c_1,\ldots,c_k$ be children of N in $T_{A,D}$ and let $ann(c_j) = [i_{j,1},\ldots,c_{j,\{N\}}]$
      Then, the node n_i in D has the following children:
      $i_{1,j}$ children labeled by c_1
      …
      $i_{k,j}$ children labeled by c_k

We assume that the compressor's annotator annotates only *dirty* nodes, and at initialization, ann(s) =[-1], for each such node.
To describe the details of the annotator, we use the following notations:
- given ann(s) =$[i_1,\ldots,i_k]$, we set last(s) = i_k
- we define ann(s) += i to be:
  If i==-1 then append "-1" to the sequence ann(s)
  If i==0 and last(ann(s)) == -1, then last(ann(s)) becomes 0 (i.e. here "-1+0=0")
  If i==1 and last(ann(s)) == -1, then last(ann(s)) becomes 1 (i.e. here "-1+1=1")
  Otherwise, increment last(ann(s)) by i.
- by child(s) we mean the "next-to-be-served" (in SAX order) child of s.

In each step of the annotator, there will be the "current" element d in D and node s in T_S, which are "synchronized"; i.e. they have the same tag path. The algorithm given below considers two cases, depending on whether we move down or up. We say that sync(d) == s, if d and s are synchronized. When we go down in D, from d to d1, and s is current in S_T, we have to find s1 in S_T, such as sync(d1)==s1
The two parts of the annotator, respectively going down (in D), and going up, are described below (both parts share a global variable temp).

1. Moving down from d to d1 (which now becomes current in D). The invariant here is: child(s)!=NULL

```
s1 = (temp == null) ? s.LeftChild : temp;
//initialize s1 to the first child of s or to temp - which is the last child visited
```

```
        while(1) { // d1 is current in D, s is current in S, need to find synchronized s1
                if(sync(d1)==s1) {
                        ann(s1) += 1;
                        break;
                }
        // take care of the subtree rooted at s1
        if(s1 != temp) ann(s1) += 0;
        s1 = s1.rightSibling            //get the next child of s
}
temp = NULL;
s = s1; // synchronized
```

2. Moving up from d to d1 (which now becomes current in D).

```
// take care of remaining children of s
if(temp==NULL) t = s.LeftMostChild; else t = temp.RightSibling;
for(; t != NULL; t = t. RightSibling;)
        ann(t) += 0;
for every child t of current s
        ann(t) += -1;
temp = s;
s = parent(s);
```

5. XSAQCT: Schema-free XML Queryable Compressor

In this section, we describe a process of building an annotated document tree, given an input document D, but with no knowledge of the grammar for D.

5.1. Building an Annotated Document Tree

We will say that a document D has a *cycle* if there exists a node n in D such that there are two children x and y of n, which satisfy this condition: x < y and y < x (here, "<" denotes the document order). If there are cycles, then add in $T_{A,D}$ a "dummy tag name", here denoted by $, which will be used to avoid cycles. The document tree may have dummy nodes, and if so they will be removed by the decompressor to recreate the original document.

Example 5.1. Document D (which has cycles on x and y) is shown in Figure 5.1.

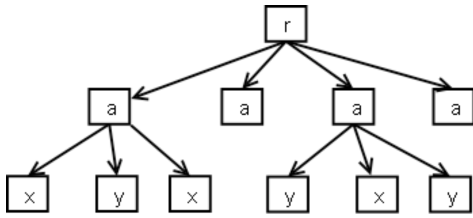

Figure 5.1. Sample document D

The annotated document tree (with dummy nodes) is shown in Figure 5.2

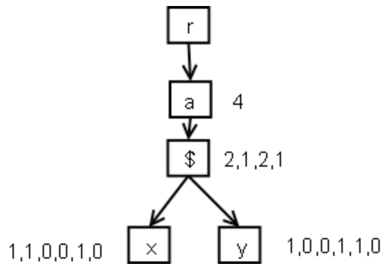

Figure 5.2. The annotated document tree.

The restored document (with the dummy nodes) is shown in Figure 5.3.

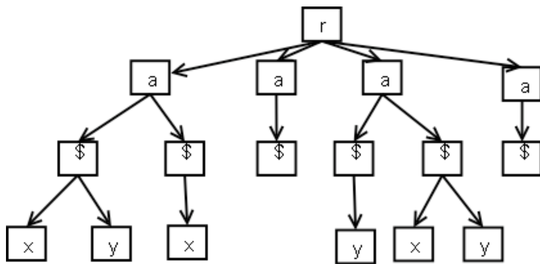

Figure 5.3. Restored document D
□

To describe details of the process of building the annotated document tree, we use the following notations:

1) ann($)+=1 means: if the annotation of $ ends with "," then append "1,"; otherwise append ","
2) ann(x)+= 1 (for another annotation) means: if this annotation ends with "," then append "0,"; otherwise append ","
3) There is a table T, each row has 3 entries: a full path, a graph associated with this path, as in the previous description, (possibly one node of this graph is "current" – see below), and an annotation for $ (this entry may be empty)
4) "close(absolute path p)" means: for each node x in the graph associated with p perform ann(x)+=1, and also if path p has a non-empty annotation for "$" then perform ann($)+=1

5) "cycle(x)" means that we are considering the node x and adding x to the graph would create a cycle (e.g. if we have a graph: a← b and we want to add a this would create a cycle a←b←a)

Algorithm 5.1
Input: An XML document D.
Output: An annotated document tree $T_{A,D}$
Method:

1) Going up from the node x to y: if x was the last (rightmost) child of y and so the next action would be going up to the parent of y, then close(x) and unset the current node in the graph
2) Going down to node x:

   - try to add x to the appropriate graph (see example below)
   - if a cycle would be created then close(x) (see 4) above), then add 1 to ann(x), and increment by 1 the annotation of $ (if such annotation does not exist or it ends with ",", then create it and initialize to 2)
   - if no cycle would be created, then add x to the graph (a new node, or just increment the annotation of existing x), and make it current node in the graph
3) After completion, check annotations and add leading 0's for regular nodes and 1's for dummies (i.e. $'s).

Example 5.2. Here, we use indices for explanations, e.g. a1 is just a.

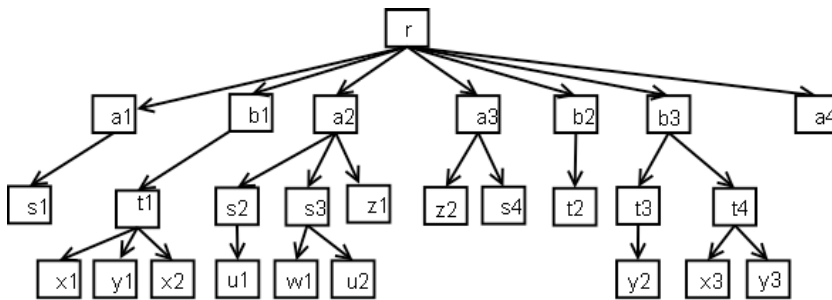

Figure 5.4. Sample document D

Table 5.1 shows all major steps in creating the annotated document tree for the document D from Figure 5.4; for each step there is a description (if needed) saying which node of the document is encountered during the SAX traversal, wherever there is a change then a path and its graph (the graph is not shown if it is empty), and the annotation of $ if it exists. The current node is underlined. Entries for leaves (where graphs are empty) are omitted. When appropriate, we show below the path and the graph; e.g. (/r a[1]) indicates the graph consisting of a single node a, annotated by 1, for the path "/r"). Only the paths/graphs that have changed from the previous step are shown. Sometimes "empty graphs" are omitted.

Table 5.1. Trace of the execution of Algorithm 5.1

1	Root r	Graph /r	
2	a1	(/r, a[1]) (/r/a, empty)	
3	s1	(/r/a, s[1])	
4	Go up to a1; close(/r/a), unset current	(/r/a, s[1,]	
5	Go to b1, add a new node b to the graph for /r and an edge between b and a	(/r, a[1]<- b[1])	
6	Go to t1	(r/b/, t[1])	
7	Go to x1	(/r/b/t, x[1])	
8	Go to y1; try to add an edge between y and x (because x is current)	(/r/b/t, x[1]<-y[1])	
9	Go to x2: this would have created a cycle. Use a rule for a cycle (above)	(/r/b/t, x[1,1]<-y[1,])	$[2]
10	Go up to t1, no occurrence of y Close /r/b/t: no current anymore.	(/r/b/t, x[1,1,]<-y[1,0,])	$[2,]
11	Go to b1, close /r/b/	(/r/b, t[1,])	
12	a2: cycle	(/r, a[1,1]<-b[1,])	$[2]
13	s2	(/r/a, s[1,1])	
14	u1	(/r/a/s, u[1])	
15	Go up to s2: close	(/r/a/s, u[1,])	
16	s3:	(/r/a, s[1,2])	
17	w1: the graph consists of 2 isolated nodes (because it had no current before)	(/r/a/s, u[1,] w[1])	
18	u2, no cycle	(/r/a/s, w[1]<-u[1,1])	
19	s3: close	(/r/a/s, w[1,]<-u[1,1,])	
20	z1	(/r/a, s[1,2]<-z[1])	
21	a2: close	(/r/a, s[1,2,]<-z[1,])	
22	a3	(/r, a[1,2]<-b[1,])	$[2]
23	z2	(/r/a, s[1,2,]<-z[1,1])	
24	s4: cycle	(/r/a, s[1,2,0,1]<-z[1,1,])	$[2]
25	Close /r/a/s	(/r/a/s, w[1,0,]<-u[1,1,0,])	
26	Up to a3; close	(/r/a, s[1,2,0,1,]<-z[1,1,0,])	$[2,]
27	b2:	(/r, a[1,2]<-b[1,1])	$[2]
28	t2:	(/r/b, t[1,1])	
29	Up to b2: close	(/r/b/t, x[1,1,0,]<-y[1,0,0,])	$[2,1,]
30	Up: close	(/r/b, t[1,1,])	
31	b3	(/r, a[1,2]<-b[1,2])	$[2]
32	t3	(/r/b/, t[1,1,1])	
33	y2	(/r/b/t, x[1,1,0,]<- y[1,0,0,1])	$[2,1,1]

34	Up: close	(/r/b/t, x[1,1,0,0,]<-y[1,0,0,1,])	$[2,1,1,]
35	t4	(/r/b, t[1,1,2])	
36	x3	(/r/b/t, x[1,1,0,0,1]<-y[1,0,0,1,])	$[2,1,1,]
37	y3	(/r/b/t, x[1,1,0,0,1]<-y[1,0,0,1,1])	$[2,1,1,]
38	Up to t4: close	(/r/b/t,x[1,1,0,0,1,]<-y[1,0,0,1,1,])	$[2,1,1,1,]
39	Up to b3: close	(/r/b, t[1,1,2,])	
40	a4: cycle	(/r, a[1,2,1]<-b[1,2,])	$[3]
41	Finish: close /r/a and then /r	(/r/a, s[1,2,0,1,0,]<-z[1,1,0,0,]) (/r, a[1,2,1,]<-b[1,2,0,])	$[2,1,] $[3,]

Note: we will add leading 0's and 1's when creating the annotated document tree in order to make sure that the number of positions in the annotations is correct. The annotated document tree created from the above table is shown in Figure 5.5.

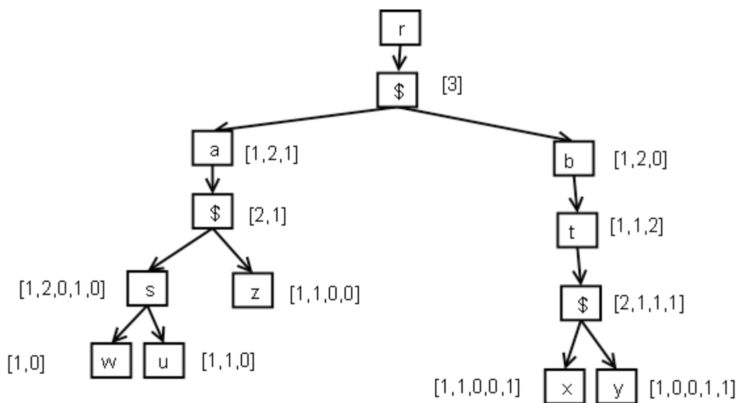

Figure 5.5. The incomplete annotated document tree

The annotated document tree in which leading 0's are added to node annotations and leading 1's are added to $'s annotations is shown in Figure 5.6.

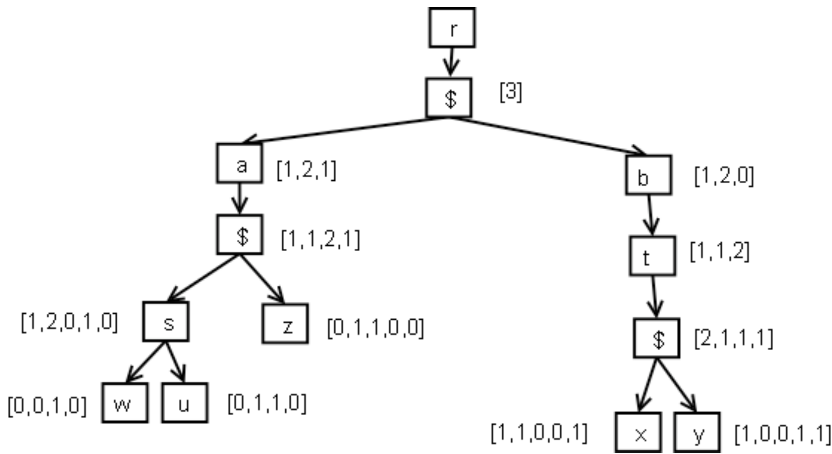

Figure 5.6. The complete annotated document tree

The restored document tree obtained from the complete annotated tree, in which dummy nodes are not removed is shown in Figure 5.7.

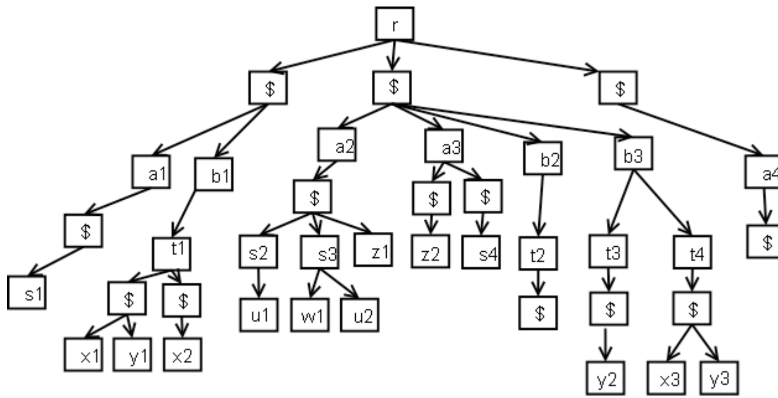

Figure 5.7. The restored document tree (dummy nodes are not removed).

It is easy to see that removing dummy nodes from the tree shown in Figure 5.7 will produce the original document tree shown in Figure 5.4.
□

5.2. Compression and Decompression

The decompressor has the following *logical* passes (the actual implementation is different, see below):

- data decompressor to restore Seq
- reannotator, re-ann() which re-annotates the skeleton tree T_D to restore $T_{A,D}$
- restorer, which uses $T_{A,D}$ to output the decompressed file

Below, we provide two versions of the decompressor; one, which will be geared towards increasing decompression time, possibly at the expense of compression

rate, and the other, which will provide faster decompression but may increase the compression rate.

5.2.3 Version I: Higher Decompression Rate

In this version, the annotated tree is first re-annotated, and then used to output the restored document.

Reannotator

Global variable, called number, initialized to 1.

```
re-ann(SkeletonTreeNode current) {
        for every child c of current {
                if(clean(c))
                        annotate c with "number" of 1's;
                else { // dirty c
                        fetch "number" digits from Seq and
                                store into the sequence "els";
                        annotate c with "els";
                        number = sum of all digits in "els";
                }
                re-ann(c);
        }
        number = sum of all digits in the annotation of n;
}
```

Restorer

```
        re-ann(root of SkeletonTreeNode);
        output <tag of the root>
        d-dfs(root of SkeletonTreeNode);
        output <\end of tag for the root>
```

where d-dfs() is described below.
Notations:

- ann(n): first digit in the annotation of the node n
- chop(n): remove the first digit in the annotation of n (always 0)
- dec(n): decrement by one the first digit in the annotation of n (never 0)
- LC(n) and RS(n): respectively the leftmost child and right sibling of n

```
d-dfs(SkeletonTreeNode c) {
        Node n;
        n= LC(c);
```

```
        while(n <> 0) {
                if(ann(n)>0) {
                        output "<" + tag_of_n + ">"
                        d-dfs(c);
                } else
                        chop(n);
                n = RS(n);
        }
        dec(c);
        output "<\" + tag_of_c + ">"
        if(ann(c)==0) chop(c);
        else {
                output "<" + tag_of_c + ">"
                d-dfs(c);
        }
}
```

5.2.4 Version II: Faster Decompression Time

The above restorer uses both tail and non-tail recursion, with tail recursion easy to remove. Below, we show a decompressor, which no longer requires that the skeleton tree to be re-annotated; instead we assume that the tree stores "pointers" to the complete annotation stream, i.e. ann(c) points to the first integer in the annotation stream. Therefore, instead of executing two separate passes; first re-annotating and then restoring, here we restore the original document in a single pass.

To remove recursion, we maintain a stack storing pairs of nodes (c,n), and assume we have three stack operations; empty(), push(c,n), (c,n)=pop(). Our tests showed that a non-recursive version executed on files from the Wratislavia corpus were between 10 and 40 percent faster that the recursive version.

```
int b; // global flag;
annotate(SkeletonTreeNode c) {
        Node n;
        while(1) { // simulate recursion
                b = 0;
                n= LC(c);
                while(1) {  // simulate returning from non-tail recursion
                        while(n <> 0) {
                                if(ann(n)>0) {
                                        output "<" + tag_of_n + ">"
                                        push(c,n);
                                        b = 1; // will jump
                                        c = n;
```

```
                        break;
                } else
                        chop(n);
                // returning from recursion here
                n = RS(n);
            }
            if(b == 1) break;  // to go to the outer loop and start all over
            dec(c);
            output "<\" + tag_of_c + ">"
            if(ann(c)==0) chop(c);
            else {
                output "<" + tag_of_c + ">"
                break;
            }
            if(empty()) return; // done
            (c,n) = pop();
            n = RS(n);
        }
    }
}
```

Note that the above code modifies annotations using functions chop() and dec(), but our implementation maintains counters, and does not modify the annotation sequence (for more details, see Section 6.)

6. Structure of C_D and Containers

In this section, we first describe the details of the compressed document C_D, and then various containers stored in C_D.

6.1 Compressed Document

The compressed document C_D starts with a one-byte long header, used to specify various options used for compression, querying and decompression, and is described below.
For both SXSAQCT and XSAQCT, the header provides the following information, as to whether:

1. labels of the annotation tree are stripped from the annotation tree, and stored in a separate (compressed) container, or stored together with the (compressed) annotation tree
2. offsets are absolute, or they are relative to the previous offset
3. the annotation tree is combined with the annotations and then compressed, or separate containers are created and then compressed
4. integer numbers are represented using variable-size-integers, VSIs, or they are

stored in four bytes

5. Text and attribute value containers are stored in a dfs-order (for the annotated document tree), or each node of the annotated document tree stores the information as to where its container is located in the compressed document

Details of the header file are as follows (below, S stands for a schema file and A stands for the annotation sequence, T_S stands for a schema tree and T_D for a skeleton tree; the corresponding bit is set to 1 if the information given in the second row is provided, and it is set to 0 otherwise):

bit 0	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7
S stored or shared	T_S stored	T_D stored	labels stripped	offsets absolute	A & T_D merged	VLI integers	containers are in dfs-order

In more details, the first two bits have the following interpretation:

- 10, if S is stored in C_D and T_S is not stored in C_D (in this case bit 2 has be to set to 0, to indicate that T_D is not stored)
- 11, if S is not stored in C_D but its URI is stored there, and T_S is not stored in C_D (in this case bit 2 has be to set to 0, to indicate that T_D is not stored)
- 01, if neither S nor its URI are stored in C_D, and T_S is stored in C_D (in this case bit 2 has be to set to 0, to indicate that T_D is not stored)
- 00, if neither S nor its URI are stored in C_D, and T_S is not stored in C_D (in this case bit 2 has be to set to 1, to indicate that T_D is stored)

Note that the following 20 bit combinations are currently unused and left for the future allocations (below, x stands for one or more "don't care" bits):

1. Four combinations of the form 101x, 111x, 011x, and 000x
2. Sixteen combinations; four of the form 100x00x, 100x10x, 100x11x, 100x01x, then four more similar combinations with respectively 11, 01 and 00 in front (if T_D is not stored then the information about labels and merging are irrelevant)

These 20 combinations are used to represent:

- 8 possible back-end compressors for annotations
- 9 possible back-end compressors to compress T_D (one of these combinations indicates no compression)

The header is followed by (the information given below may or may not appear depending on the values stored in the header; all the containers are compressed):

1. One of: the URI, the schema, or the schema tree.
2. Annotation container
3. The annotated tree (see below for more details)
4. The container storing labels of this tree
5. A sequence of compressed containers, storing text and attribute values; each container is preceded by its length.

There is a special byte, called a node mask (NM) added to each node of $T_{A,D}$, which represents the following values (which may or may not be significant, depending on the value of the C_D header):
bit 0: 0 clean, 1 dirty
bit 1: (used only for clean nodes, which have all annotations equal), 0 if VLI representing each annotation is not equal to 1, and in this case is stored following NM, and 1 if each annotation is equal to 1
bit 2: 0 regular , 1 dummy
bit 3: 0 attribute, 1 element
bit 4: 0 if no mixed content, 1 if mixed
bits 5,6,7: 8 possible data compressors

Note that the following bit combinations are currently unused and left for the future allocations:

1. Combinations for attributes of the form xxx0x
2. Combinations for dummy elements of the form xx1x

The entire tree $T_{A,D}$ is implemented using the leftmost-child right-sibling representation in the array. Each node of $T_{A,D}$ stores the following information:

- leftmost child
- right sibling

6.2 Containers

During the compression, when the original document is parsed for the first time, the implementation maintains buffers, which are flushed to containers when they become full. (In the future, we will experiment with various sizes of buffers.)
Figure 3.6 shows homogenous containers for nodes with mixed contents may require scanning the annotation container in order to answer queries such as /a/b[2]/*. To avoid this scanning process and facilitate such queries, containers may be further sub-divided into sub-containers. Here, we will consider two possibilities:

- fixed sub-containers created by the compressor (at the expense of storing the additional information in the annotation tree)
- creating sub-containers on demand, during the querying process and storing this information in the cache.

7. Querying

Since at the time of writing this report, our query processor was under development, here we provide only some ideas about its design.
We assume *lazy initialization* of the compressed document C_D; i.e. when the user performs a query for the first time, the initialization process takes place:

1) The skeleton tree and the container storing labels are retrieved from C_D, decompressed and used to restore T_D, in internal memory, in a form of an array;
2) As we mentioned in Section 6, the tree $T_{A,D}$ is implemented using the leftmost-child right-sibling representation in the array. The array used to represent tree $T_{A,D}$ is now *complemented* by the second parallel array, which, for each node of this tree stores the following data:
   a) integer values representing the *left-sibling* and the *parent* (to facilitate use of various axes for queries)
   b) one-bye long node masks
   c) three-bytes long relative offsets into annotations

Note that other containers storing text and attributes values are not decompressed during the initialization.
We assume that the compressor creates a separate container for *similar* paths, and each container has a header storing its compressed length (these containers are then concatenated). We also assume that each container is compressed (by the standard data compressor, appropriate for the kind of data used in this container) separately.

Example 6.1
Here, we consider querying for semi-leaves for the document D shown in Figure 6.1, with the containers (stored consecutively in the dfs-order order) as shown in Figure 6.2.

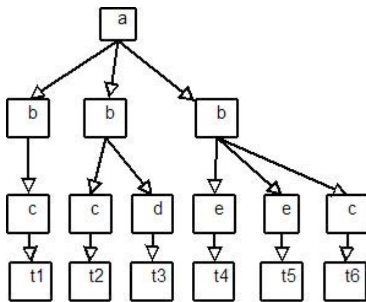

Figure 6.1. Sample document D

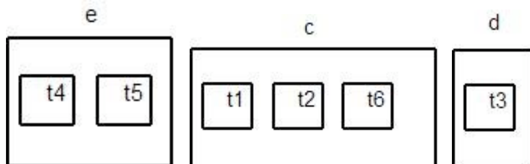

Figure 6.2. Text containers for D

Now, consider the following query: /a/b/c. From the skeleton tree T_D shown in Figure 6.2, we know that the container for “c” is the second one, and so retrieve it, and decompress.

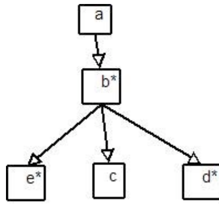

Figure 6.3. Skeleton tree for D □

8. Results of Experiments with XSAQCT and SXSAQCT

The initial implementation of our compressors was completed using Java and Xerces. The results of our experiments are reported in Table 8.1.

Table 8.1 Results of our experiments

	uwm	shakespeare	lineitem	SwissProt	enwikinews	enwikibooks	dblp
S_T Nodes	43	155	37	394	50	50	313
Dummy Nodes	0	9	0	2	0	0	10
S_T size (unc)	384	1,051	383	3,495	467	467	2,248
S_T size (com)	211	309	200	779	251	251	570
Compr. time	1.246	3.651	4.426	29.836	7.652	23.279	39.85
Init PAQ compr %	0. 0.00315	0.0002643	0.000235	0.000188	0.000003783	0.0000009829	0.0000550

9. Conclusion and Future Work

In this paper, we described our current work on two new queryable XML compressors. While this work is in its early stage, the design of these compressor and the results of our experiments indicate that they may successfully compete with other known queryable XML compressors. Our future work includes:

- Rewriting the code using C++ rather than Java
- Handling recursive schema
- Analyzing containers to determine the best back-end data compressor
- Building a complete query processor

References

[Skibinski] P. Skibiński, Sz. Grabowski, and J. Swacha - "Effective asymmetric XML compression", Software - Practice & Experience, 2007/2008.