Compression of Probabilistic XML documents

Irma Veldman
i.e.veldman@student.utwente.nl

July 9, 2009

Abstract

Probabilistic XML (PXML) files resulting from data integration can become extremely large, which is undesired.

For XML there are several techniques available to compress the document and since probabilistic XML is in fact (a special form of) XML, it might benefit from these methods even more.

In this research we search for compression mechanisms that are available for XML and implement one of them to customize it with respect to the properties of probabilistic XML.

Experiments show that there is no significant improvement for combinations of traditional mechanisms with techniques that are specially designed for probabilistic XML.

1 Introduction

Probabilistic XML (Pxml) is XML that contains uncertainty in the data. This uncertainty can be caused by the integration of two or more XML documents. During the integration of documents, conflicts can arise. It is wise not to resolve these conflicts at integration time, because this requires a huge amount of user effort. Instead, the uncertainty can be left in the document and resolve conflicts when they emerge , i.e. at query time.

In the field of probabilistic XML some good solutions have been achieved with respect to data representation and efficient querying of the uncertain data, as in [14]. The authors use an example of integration that we will use throughout the paper, too. The example consists of two small address books, each containing a record of a person named `John`, whose phone number is `1111` in one address book and `2222` in the other. Integrating these two address books will result in a conflict. In the real world different situations could be possible:

- Both records refer to the same person named `John`, but one of the phone numbers is wrong.
- Both records refer to a different person named `John` and for each person the phone number is correct.

A rule engine could assign probability values to these different situations and during integration these possible situations are then turned into a PXML document, which represents these possibilities. This way the uncertainty is kept in the data and after the integration the probabilistic XML document could look like the PXML tree in Figure 1. This figure is originally from [14].

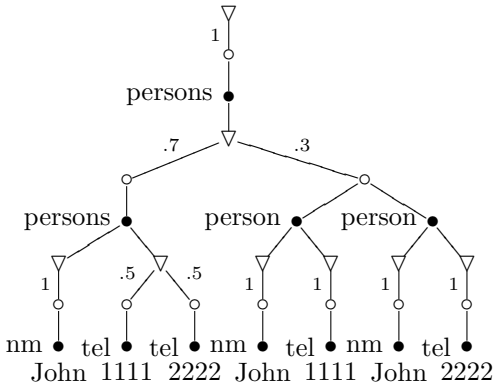

Figure 1: Example probabilistic XML tree.

The ▽ nodes represent the probability nodes. The ○ nodes represent the possibility nodes. They have an associated probability value. This value lies within the range $\langle 0.0, 1.0]$. The actual value of such a possibility is determined by a rule engine. The • nodes represent normal XML nodes.

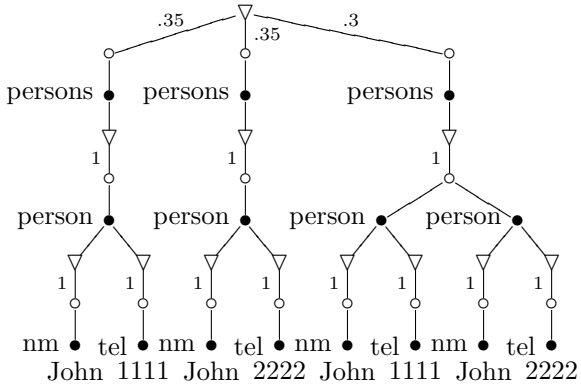

Figure 2: Possible world style (PWS) tree of Figure 1.

The next step is to query the data. This is what [14] says about querying uncertain data:

> "*Uncertainty can be treated as having more than one possible instantiation describing a particular real world object. Choosing one possible instantiation, or* possibility *for short, for each real world object, results in a possible world. Analogous to the notion of parallel universes, all possible worlds co-exist in the database and a query should, therefore, be evaluated in every possible world separately.*"

Thus we need to construct possible worlds. Figure 2 shows us the 3 possible worlds in case of the example.

Unfortunately, this *Possible World Style* (PWS) comes with a major drawback, which does not emerge from this example because it is too small. Imagine the integration of address books with over 100 records each. With n ($n < 100$) conflicting records, each with 3 possible real world situations, the PWS of the document could grow a factor 3^n.

Luckily, current query engines are able to handle compact representations of the PWS. In these compact representations possibilities are pushed down to lower levels in the tree and probability and possibility nodes that do not provide extra information are removed, see Figure 3 and section 4.

In this research we will look for methods to further compact PXML documents in order to decrease the amount of space needed and hence improve performance.

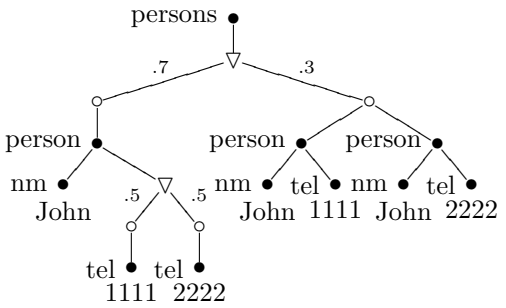

Figure 3: Same tree as Figure 1, but without the probability and possibility nodes that give no extra information.

2 Research

In the field of XML, a lot of effort is spent on the compression of XML documents. One of the methods mentioned in the literature is the use of Directed A-cyclic Graphs (DAGs). Fundamental theory here is the sharing of identical subtrees. In an XML tree of a document, there can exist multiple identical subtrees. All subsequent subtrees can be replaced by a link to the first occurrence of this identical subtree. After this event the XML tree is no longer a tree, but a DAG. This is an important difference, because it effects how queries should be evaluated.

Since probabilistic XML is in fact XML (e.g. it suffices all XML constraints, it just adds some constraints), these compression techniques should work on PXML documents as well. However, since PXML is a special form of XML, there might be chances of improving the compression even more, by taking these special properties into account. This will be the main focus of this research.

2.1 Research Question

We can translate the idea mentioned above into the following research question:

> *Can we improve a standard compression technique for XML by taking into account the special properties of PXML in order to improve the compression ratio for PXML documents?*

2.2 Research Approach

In order to give an answer to the research question the following steps are taken:

- First, we will explore which compression techniques are proposed for XML in the literature. (Section 3)
- We explore possible compression techniques for probabilistic XML. (Section 4)
- From the techniques found in the literature we will implement one of them in a prototype. The prototype will be expanded with existing methods to compact the PXML document. (Section 5)
- A representative measure has to be determined to be able to calculate the compression ratio. (Section 6)
- Experiments are done to measure the amount of compression with and without the expansion of the prototype. (Section 6)
- The results of the experiments will be analyzed and conclusions are drawn. (Section 8)

3 XML Compression

XML is a standard data format which is gaining popularity. It is used for exchanging data between applications on computers and mobile devices. The authors of [8] foresee that in the future massive amounts of XML data will be generated and exchanged.

Unfortunately, the verbosity of XML causes the documents to be very large. This is a serious drawback of XML and finding a compression technique for the reduction of the size of XML documents is a hot issue these days. Several compression techniques have been proposed, each having its own characteristics. We will explore these techniques in the rest of this section. The list of techniques mentioned here is not exhaustive. We selected the most recent and/or discriminative techniques.

There are several ways to distinguish the different compression mechanisms. In the comparative study of Ng et. al. [7] the authors chose to categorize the compression techniques into queriable versus unqueriable techniques. Queriable techniques come with the important feature that they can be queried directly on the compressed data, but unfortunately do not perform as well in terms of compression ratio and execution time. Unqueriable techniques need to fully decompress their data before it can be queried again, but they can achieve a much higher compression ratio.

In our study of XML compression techniques we will also group the different compressors by their (in)ability of supporting queries. In section 3.3 we will pay special attention to the compression technique that served as the basis for our own prototype.

3.1 Unqueriable compression techniques

From the unqueriable compression techniques studied in [7], XMill [5] and XMLPPM [3] do not use any kind of schema. XMill simply compresses the skeleton of the document using a dictionary encoding approach for the tag and attribute names. The data values are grouped into containers based on the tag or attribute they belong to. With some user intervention, the grouping can become more efficient. Then the skeleton and the data containers are compressed using Gzip.

XMLPPM produces an encoded SAX event stream. The SAX event stream of the input document is processed by a set of four PPM coding models. Each model is responsible for the encoding of special parts of the document, for example attributes, or strings.

Both Millau [10] and SCA (Structure Compression Algorithm) [4] use DTDs for the compression of XML documents. They encode the document information that cannot be inferred from the document associated DTD, which includes data values and structural information. Millau parses the DTD and the document simultaneously, which results in a structure stream and a content stream. The content stream will be compressed in another phase of the algorithm. SCA differs from Millau in the way the structure stream is created. Millau first produces a parse tree that represents the DTD and the document simultaneously. This parse tree will be transformed into a pruning tree, remaining only the structure nodes that cannot be inferred from the DTD. The data values are separated and outputted to a content stream, which will be en-

coded by generic compressor. Both methods suffer from huge memory consumption.

3.2 Queriable compression techniques

XGrind [11] and XPress [6] are queriable compression techniques that adopt a homomorphic transformation, which means that the structure and semantics of the XML document are preserved. This enables the document to be parsed as any other XML document. As with XMill, XGrind uses a dictionary encoding approach for the tag and attribute names. The data values are encoded by Huffman encoding (for the non-enumerated attribute values and the PCDATAs) or binary encoded (for the enumerated attribute values).

XPress [6] uses a reverse arithmetic encoding scheme for the encoding of the skeleton. This method encodes not only the tag name, but also the tree path to this tag. Such a tree path is modeled as a real number interval in the range $[0.0, 1.0\rangle$ that satisfies the suffix containment property. This means that if an element path P is a suffix of an element path Q, the interval that represents P should contain the interval of Q. XPress can automatically determine the type of a data value and hence apply the proper compression for it. Besides, XPress also supports query updates directly on the compressed data.

XCQ [9] is a compression technique that also uses DTDs as with Millau and SCA. It uses the DTD tree and a SAX parser to construct a structure stream that contains all information that cannot be inferred from the DTD on itself. Simultaneously, the data will be grouped in containers based on their tree paths. These containers are divided into blocks. The blocks are compressed and get a BSS (Block Statistics Signature) index. A cost model can decide to group the blocks into clusters or even multiple clusters with a CSS (Cluster Statistics Signature) or MSS (Multiple cluster Statistics Signature). During querying, only blocks or clusters that contain the relevant information need to be decompressed.

Another approach for the compression of the skeleton of an XML document is the use of DAGs (Directed Acyclic Graphs). This technique is based on the sharing of common subtrees and is applied in [1]. The compressed document is still queriable and results can be returned in compressed form to serve as an input for another query.

3.3 BPLEX

We pay some special attention to the compression technique, called BPLEX, of [2], because it is the basis for our own prototype. It takes the idea of transforming the XML tree into a DAG somewhat further. It is based on the sharing of common subgraphs instead of common subtrees. This makes it possible to share parts of a subtree instead of complete subtrees, which increases the sharing opportunities.

XML trees can be expressed as grammars. The minimal unique DAG can also be seen as the minimal regular tree grammar that generates the tree. A generalization of the sharing of subtrees is the sharing of arbitrary patterns, i.e., connected subgraphs of a tree. A sharing graph can be seen as a context-free (cf) tree grammar.

A small example of this (used in [2]) is a tree *c(c(a,a),c(a,a))*. Listing 1 shows us the XML of this example. The corresponding tree can be seen in 4(a). The minimal DAG for this tree can be described by a minimal regular tree grammar consisting of the following productions: $S \rightarrow c(V,V)$, $V \rightarrow c(W,W)$ and $W \rightarrow a$. The DAG is illustrated in Figure 4(b).

Listing 1: XML code of the example

```
<c>
  <c>
    <a/>
    <a/>
  </c>
  <c>
    <a/>
    <a/>
  </c>
</c>
```

We illustrate the idea of a sharing graph in the next example, also from [2]. We take the tree *c(c(a,a),d(c(a,a),c(c(a,a),d(c(a,a),c(a,a)))))* which is depicted in Figure 5(a). In this tree, there is a pattern (5(b)) that is repeated. Because different subtrees are hanging underneath this pattern would be useless for building a DAG.

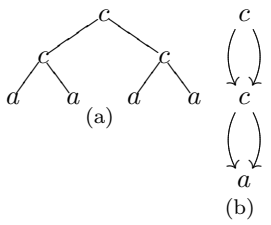

Figure 4: Example XML tree (a) and its corresponding minimal DAG (b).

However, with the introduction of formal parameters, we can share this subgraph. The resulting grammar would have the following productions: $S \rightarrow B(B(C))$, $B(y_1) \rightarrow c(C,d(C,y_1))$, $C \rightarrow c(A,A)$ and $A \rightarrow a$. This context-free grammar is also called a *straight-line* (SL) grammar.

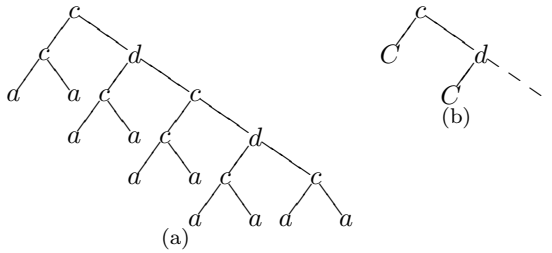

Figure 5: Example XML tree (a) with the pattern $C \rightarrow c(A,A)$ and $A \rightarrow a$ (b) that cannot be used to turn into a DAG.

The sharing of the pattern is depicted in Figure 6(b). We see that the most upper c has two special incoming edges. These special incoming edges are recognizable by the $\perp$ symbol at the end of the edge. This means that the subtree is shared from here.

Walking through the tree, we arrive at this c from the incoming edge marked with a 1. This number at the end of the edge means that whenever a choice has to be made between two or more outgoing edges belonging to a choice, recognizable by the $\perp$ symbol at the start of the edge, you have to choose the one with the same number. In this case this outgoing edge itself is again a special incoming edge, marked with a 2, meaning that the next time you come across a choice point, you have to choose the other special outgoing edge. The other 'normal' outgoing edge from node d is not marked, hence it is shared. The numbers alongside the edges represent the formal parameters in the grammar.

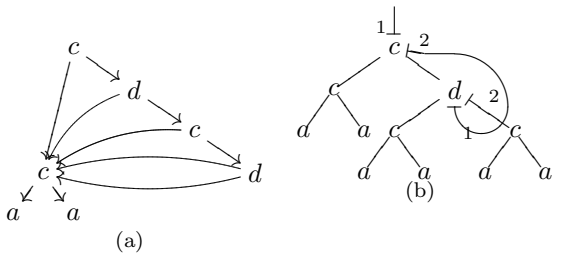

Figure 6: The DAG created from 5 (left) and the plexed version (right).

In Figure 6 we can also see what difference it makes between creating a DAG from the tree in Figure 5 and plexing it. The original tree has 19 nodes. The DAG has already reduced this to 7 nodes. Obviously, on the plexed tree the $c(a,a)$-subtrees can also be shared, even completely. However, we did not do that for simplicity. However, when we would have done that, another 6 nodes disappear and we are left with only 5 nodes, which is less than the DAG variant.

The BPLEX algorithm presented in [2] stands for *bottom-up multiplexing* and takes as input an SL regular tree grammar G and three parameters. The algorithm walks along all symbols in the grammar in SL-ordering of G (which is like post order traversal in a tree). For a window with a maximum size (specified by one of the parameters) it calculates if there are matches of patterns. These matches must satisfy the maximum size of the pattern (specified by a parameter) and the maximum rank of a new pattern (also specified by another parameter). The rank of a node n is the number of child nodes. From all the matches that can be found, the maximal match is chosen and the grammar G is adjusted accordingly. The output of the algorithm is the multiplexed grammar G generating the optimized tree. This is a very short summary of what the algorithm does. For a more detailed version, see the pseudo algorithm in [2].

4 Compact PXML

PXML is a form of XML that follows some constraints to satisfy the probabilistic representation.

We follow the representation of PXML as defined in [14]. For the formal definition, we will refer to that paper. For now, we will give a short description:

- The root node is always a single probability node (▽).
- Each probability node only has possibility nodes (∘) as child nodes.
- Each possibility node comes with an attribute named *prob* and has a value v with $0 < v \leq 1$.
- The sum of all probability values of the child nodes of one probability node is 1.
- The child nodes of a possibility node are normal XML nodes (•).
- The child nodes of a XML nodes are probability nodes.

Notice that figure 1 suffices this description, as well as the PWS representation of this figure, which can be seen in figure 2. Exceptions are the text nodes which are hanging directly under XML nodes. In the sequel (unless explicitly noted), text nodes will appear directly underneath the XML nodes in diagrams, but one should interpret it as if there are a probability and possibility node in between.

As mentioned before, the PWS is a suitable representation to query upon. Unfortunately, for large documents, this PWS can become inconveniently large. Therefore, a more compact representation is desired.

4.1 Deleting redundant nodes

One step to compact the tree is the most obvious step. Notice that PWS document contains the pattern depicted in Figure 7(a) many times. Since the combination of the probability and possibility node do not add any value, we can remove them, which leaves us with just the XML nodes, depicted in Figure 7(b).

Notice that this routine applied to the XML tree in Figure 1 will produce the tree in Figure 3.

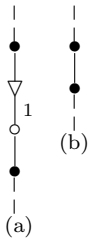

Figure 7: Pattern that is often seen in PWS documents (a) and how this can be reduced (b)

4.2 Simplification

Besides this previous simple step, there is another mechanism for reducing the number of nodes, which is a push-down mechanism for possibilities. When pushing down the possibilities to lower levels of the tree, some top nodes of different possible worlds can be shared, which decreases the number of nodes.

This routine is called simplification and can be seen as the inverse of making a document into PWS. We will illustrate this routine with an example. Then we will discuss the general idea.

For this example we take the tree in Figure 2. We will show in incremental steps how this tree can be turned into the tree of Figure 3. Note that in these trees the probability and possibility nodes between the lowest XML nodes and text nodes are visible.

We start at the first level of possibility nodes. We look for candidate nodes for merging. The first layer of possibility nodes can be merged, because they all have an XML node labeled with `persons` and this node has in all three cases only one probability node. Figure 8 will show what happens to the tree.

Then we explore the next layer of possibility nodes. We cannot merge all three of them, since the third possibility node has two XML nodes as children. The first and second possibility node however are still candidates. Unfortunately, they both have more than one probability node as children. But with a small step in between, we can change that. We can merge the two probability nodes and the two possibility nodes underneath. The two XML nodes are now children of the merged possibility node. This situation is depicted in 9. Notice that the semantics still do not

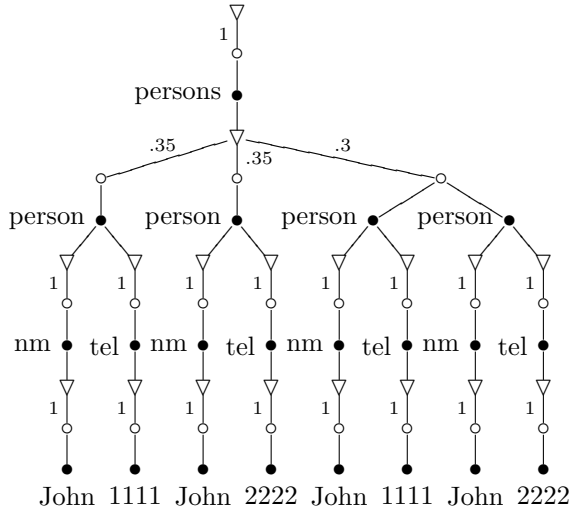

Figure 8: First iteration of simplification performed on the tree of Figure 2

.

change.

Now we have a situation similar to the first step we took, since we merged the two probability nodes under the first two XML nodes labeled `person`. Thus we can now merge the two possibility nodes (each with $prob = 0.35$ chance), the two XML nodes and the two probability nodes. This is depicted in Figure 10. Notice how the probability values of the merged possibility nodes and the possibility nodes one layer down are adjusted.

We can observe that in both subtrees of the first XML node labeled `person` there is a subtree that is completely equal to the other. Semantically this means that we have one person with either telephone number `1111` or `2222`, but we are sure that his name is `John`. Hence we can remove these subtrees, merge them and make them a certain subtree of `person`. The result of this action is depicted in Figure 11.

Finally, if we would apply the routine whereby the redundant probability and possibility nodes are removed, we end up with the tree shown in Figure 3.

We can generalize the simplification into three different transformations. The first transformation is the most simple one. It is shown in Figure 12. It means that if an XML node has two XML nodes as children, without any uncertainty, we can also ex-

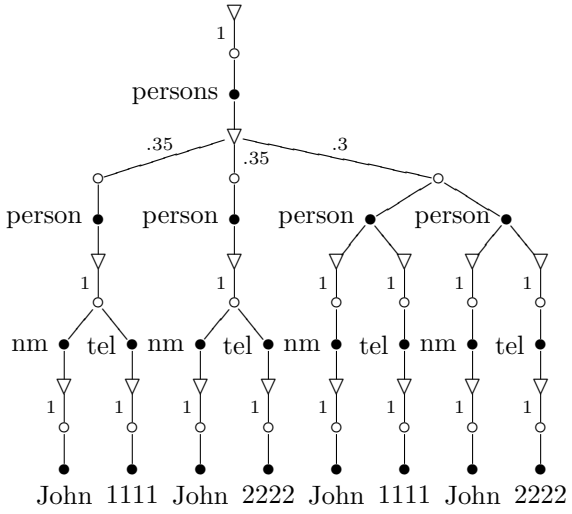

Figure 9: Second iteration of simplification performed on the tree of Figure 2

.

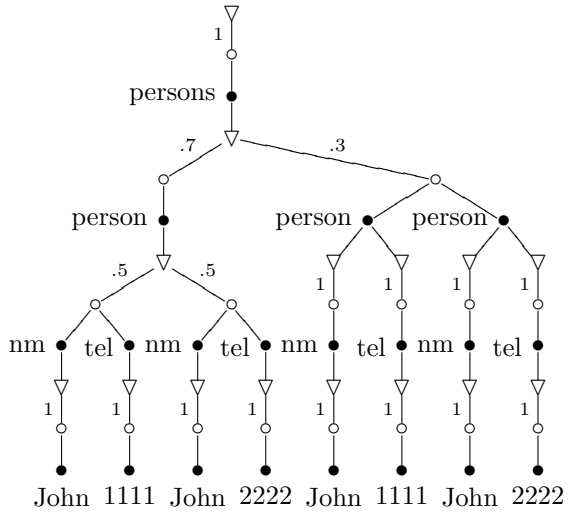

Figure 10: Third iteration of simplification performed on the tree of Figure 2

.

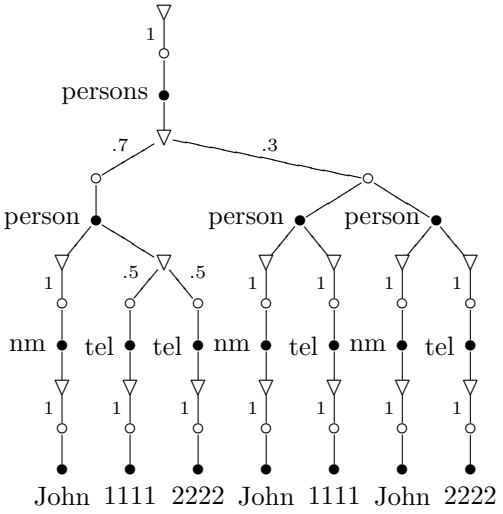

Figure 11: Fourth iteration of simplification performed on the tree of Figure 2

.

press that with only one probability and possibility node.

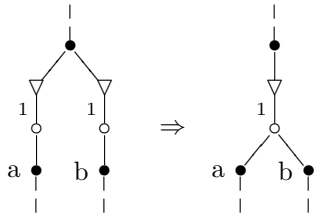

Figure 12: One of the possible transformations during simplification

.

The next possible transformation is the one shown in Figure 13. Whenever there are a number of possibility nodes (having the same parent) that each have one, identical XML node as child, which has on its turn only one probability node as a child, we can merge these nodes. The probability value of this merged possibility node is the sum of the old possibility nodes. The probability value of the possibility nodes that now are children of the merged probability node are recalculated.

Finally, the last possible transformation is shown in Figure 14. If every subtree of a probability node contains the same subtree, then all these subtrees can be removed and one of these subtrees can be placed as a sibling of the probability node. The labels b, b' and b'' mean that although the XML node might be identical, the complete subtrees rooted at these nodes are not.

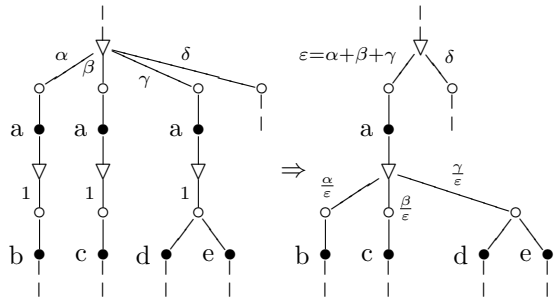

Figure 13: Fourth iteration of simplification performed on the tree of Figure 2

.

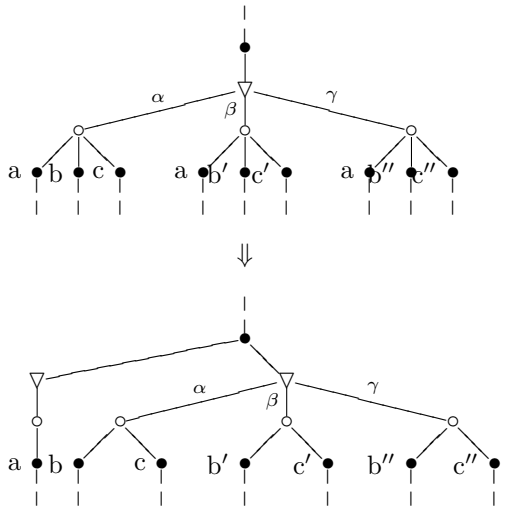

Figure 14: Fourth iteration of simplification performed on the tree of Figure 2

.

Notice that for the second and third transformation it is sometimes required to perform the first transformation to get a pattern match. It is possible to do these transformations in one single transformation, but then it is harder to process them. Besides, the first transformation reduces the number of nodes anyway, even if the patterns of the second and third transformation cannot be recognized. Hence, we have split them.

5 Prototype

To test whether a standard XML compression technique can be improved by inhabiting the special properties of PXML, we have built a prototype. In the prototype we implemented the mechanisms to compact the PXML based on the PXML-properties, to build a DAG and to multiplex it. Combinations also possible, too.

5.1 Goal

Main goal of the prototype is to measure the compression ratio of PXML documents. It should compare a conventional XML compression technique, techniques customized for PXML and combinations in terms of size. As for now, the performance of the prototype in terms of compression and decompression speed will not be part of the comparison.

5.2 Implementation

As mentioned before, the inspiration for the compression of normal XML was the BPLEX algorithm of [2]. We believe that the algorithm can also be performed on the DOM structure of the tree instead of the SL grammar and hence decided to build our own implementation of the algorithm. We also extended the basic DOM structure that is already there, to be able to function properly with this algorithm. In the sequel, we will call this algorithm PLEX.

5.2.1 Input and Output

Before we discuss the prototype in more detail, we first take a look from a black-box perspective. Here we state what we expect from an XML file that serves as input for the prototype and we decide what a compressed document should look like.

Requirements Since it is just a prototype, we do not perform a thorough check on every input file. Therefore we carefully state what we expect from an input file.

The input file a) should contain well-formed XML, b) should not contain comments, c) may contain attributes and d) should suffice the PXML format, e.g. probability and possibility nodes between the XML nodes.

Representation We have seen how the sharing of common subgraphs can be depicted in a grammar. We now need a proper representation in XML. Since we like the idea of homomorphic compression, i.e. the compressed document contains structure, we should choose a representation that in itself will be an XML document again. This requirement is important, because this enables the possibility of performing queries on the compressed document. Actually querying the compressed document however is not within the scope of this paper.

Most important is the ability of referring to another tag in the document. Therefore we need to identify each element with a unique number. We will attach this number to a tag as an attribute with name *id*. In Listing 2 the representation for the DAG from Figure 4 is shown.

Listing 2: Representation of the DAG of Figure 4

```
<c>
  <c id='0'>
    <a id='1'/>
    <ref>1</ref>
  </c>
  <ref>0</ref>
</c>
```

Secondly, we need to discriminate different subtrees. Take for instance the tree from Figure 15. In this tree, the left and the right subtree right underneath the root share all nodes except for one. Ideally we would merge these nodes, which would have been impossible in case of DAGs. We may plex the tree though. In Figure 16, we see how this would look like in a diagram of the tree. From the root, both outgoing edges are directing to the possibility node underneath. The first edge we pick is obviously the left one (since we do this in 'normal' trees as well). Seen from the possibility node, this edge is a special incoming edge marked with the number 1. Hence, if we come across a choice point for this subtree we have to choose the choice marked with a 1. For the right subtree, the same is valid for the number 2. As we can see, this choice point is just before the telephone numbers. The sharing

of the subtrees ends here and for every subtree the correct choice is made.

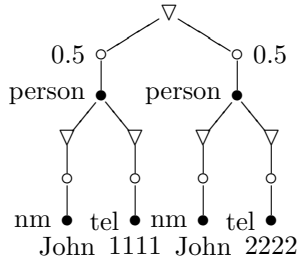

Figure 15: Example tree before plexing

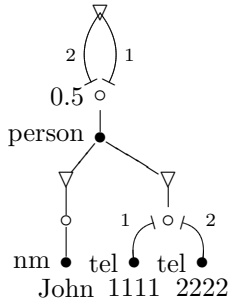

Figure 16: Example tree of Figure 15 after plexing

Important is how we can translate this into normal XML. The representation of this example is shown in Listing 3.

From this example we can see that tags that get referenced receive an *id*, which is attached to the element as an attribute (line 2). In line 17 we see a *reference* to an element with *id='1'*. This means that we can replace this *referencing* element with a copy of the subtree rooted at the element with *id='1'*.

Then we have the issue of the choice point in the tree, in this example the element `tel`. As we can see, both choices appear in the first subtree, but are marked as a choice point, e.g. *param='1'*. To make sure that the subtree starting in line 2 and the subtree starting at line 17 pick the correct choice, they are marked with an attribute *input*. This *input* means that whenever you encounter a choice point in the subtree rooted at the element labeled with an *input* you have to pick the choice that is marked with a *param* equal to your *input*. Note that, when you replace line 17 with a copy

Listing 3: Representation of the plexed tree of Figure 16

```
<prob>
  <poss prob='0.5' input='1' id='1'>
3   <person>
      <prob>
        <poss prob='1.0'>
          <nm>John</nm>
        </poss>
8     </prob>
      <prob>
        <poss prob='1.0'>
          <tel param='1'>1111</tel>
          <tel param='2'>2222</tel>
13      </poss>
      </prob>
    </person>
  </poss>
  <ref input='2'>1</ref>
18 </prob>
```

of the subtree starting at line 2, the *input* of the referencing element will override the original *input*.

Notice that this representation in itself is XML again and hence satisfies the homomorphic requirement.

5.2.2 Supported Compression

Our prototype supports several compression techniques, which is practical for experimental reasons. One may specify which compression technique is applied to the XML document. The choices are:

- Perform the simplification technique
- Remove all redundant probability and possibility nodes
- Turn the document into a DAG or plex it.

Combinations are possible as well. Notice that one cannot perform DAG and PLEX method at the same time. They are each others alternative. For every compression technique, there is an inverse method too, that makes the compression undone. Creating a DAG and the plexing of the document are discussed in more detail in the next section.

5.3 Creating a DAG

The creation of a DAG starts with the search for matching subtrees in the tree. The idea is simple: you walk through the tree in post order. A window contains the nodes you already visited. And for each node n you visit, you search for equal nodes in the *window*. For each of these nodes $matchInWindow$, you should check if the complete subtrees rooted at $matchInWindow$ and n are equal. If they do, they are a *match*. Then for each such match, it is tried to expand this match in an upward direction. So the parents should be equal and the rest of the subtree that wasn't already included in the match rooted at n and $matchInWindow$ should be equal as well. For all matches found for this node n, we can determine the optimal match, which is de match with the most equal nodes. This is the match eventually chosen to apply. Listing 4 shows the pseudo code for this algorithm.

Listing 4: The DAG algorithm in short

```
PxmlMatch[] matches;
PxmlNode window = null;
for(PxmlNode n: document) {
  PxmlMatch[] matchesForN;
  PxmlNode[] equalNodes =
    findNodesInWindow(n);
  for(equalNode e: equalNodes) {
    if(equalSubtrees(equalNode,n) {
      while(expandMatch(equalNode,n) {
        equalNode =
          equalNode.getParentNode();
        n = n.getParentNode();
      }
      matchForN.add(
        new PxmlMatch(equalNode, n));
    }
  }
  PxmlMatch maxMatch =
    getMaxMatch(matchesForN);
  matches.add(maxMatch);
  window = window + n;
  n = nextInPO(n);
}
for(PxmlMatch m: matches) {
  applyMatch(m);
}
```

The application of the matches in case of DAGs is relatively reasonably. For each match, replace the matching tree with a referencing node to the root of the original subtree to which it matches. If this original tree is already been replaced in some previous match, then we query this original tree for its new location and refer to that subtree instead. Hence, for each node in the replaced (or deleted) subtree, we need to set the new location in case another match refers to nodes in this tree.

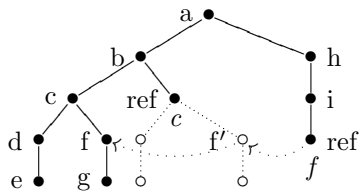

Figure 17: The child node of node i was first referring to node f', but since this node is replaced, the child node of i is now referring to node f.

Take for instance the tree in Figure 17. Node b originally had two equal subtrees. The second subtree is replaced by a referencing node to the root of the first subtree rooted at c. Now the child node of i was already marked to match with f', but this node is replaced and is no longer part of the original tree. Thus we query the new location of the node f' and discover that this is node f. We can now safely refer to f instead.

5.4 Plexing the document

Plexing a document starts the same as with the creation of a DAG. You walk through the tree in post order and compare the current node n with the current *window*. A match is found when n and a certain node in the *window* are equal and have equal subtrees. Then we try to expand this match in upward direction. This time this already succeeds if both the parents of n and $matchInWindow$ match, because no longer complete subtrees have to match. In a special copy of the subtree rooted at $parentOfN$ we mark which nodes are equal, and which are not. This will be discussed in more detail in section 5.4.1.

In case of plexing, not every match can be applied. We elaborate on that in more detail in section 5.4.2. Because not every match can be applied

straight away, we do not only determine the maximal match, but also remember other matches, in case the maximal match cannot be applied. For the maximal match, we check if it is valid to apply. If not, we try an alternative. The last three lines of pseudo code of Listing 4 (lines 24-26) can hence be replaced by the lines in Listing 5.

Listing 5: The lines that replace the last three lines of Listing 4 in case of plexing.

```
for(PxmlMatch m: matches) {
25   if(validMatch(m)) {
       applyMatch(m);
     } else {
       m = m.getAlternative();
     }
30 }
```

5.4.1 Match

As already mentioned a match consists of two links, each referring to a node that is root of a subtree (partially) equal to the other subtree. The first subtree is always the first occurrence of the subtree and will be the "original" to which the second subtree will be referring, in case of application. Besides those links, a match consists of something else. When the match is created, another tree is made. It is a sort of copy of the subtrees, marking which nodes are equal and which are not. For each equal node, a referencing node (*PxmlRefNode*) is created, referring to the node in the first subtree. Whenever there are nodes not equal, a copy of the node in the second tree is made. For all the matches that can be made for a node n while walking through the tree, the maximal match is the one with the highest number of PxmlRefNodes in the third tree. Each match has a link to the next-best match.

5.4.2 Valid Matches

Since the matches of a DAG do not change the "original" subtree, there is no need to check whether a match can really be applied. However, in case of plexing, a previous match could have changed the original subtree. Then another subtree matching this original subtree might not match anymore. So an extra check needs to be performed. We illustrate this with an example.

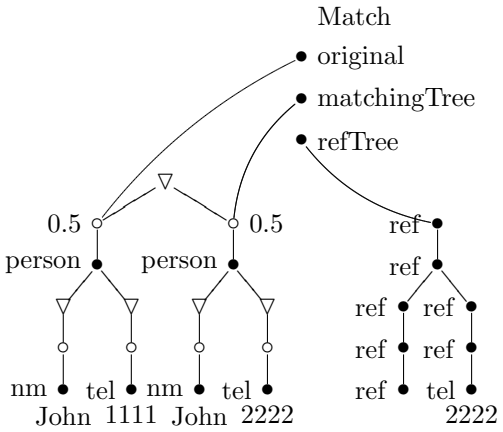

Figure 18: An example of a PxmlMatch

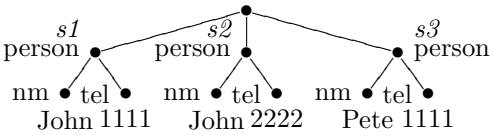

Figure 19: A simple tree with three subtrees

In this example we have the simple original tree shown in Figure 19. We have a root node with three subtrees *s1*, *s2* and *s3*. *s1* and *s2* share the name and *s1* and *s3* share a phone number. Unfortunately, we cannot apply both matches. Figure 20 shows us what happens after applying the first match. Since the name is equal in both subtrees we can share this part. The phone numbers are not equal and hence the distinction is made between the subtrees with the *input* and *param* parameters. If we would try to apply the second match, the distinction has to be made between the names, since this part of the subtrees of *s1* and *s2* are not matching. However, the name of *s1* is not only part of *s1* anymore, but has become *shared*.

Figure 21 shows us what happens if the match would have been applied anyway. We see that the name `John` receives the *param* value of the input of the original subtree *s1*, namely *1*. And `Pete` receives *param=3*. And because the phone numbers are matching, nothing has to be done for that subtree. If you would undo this compression, you

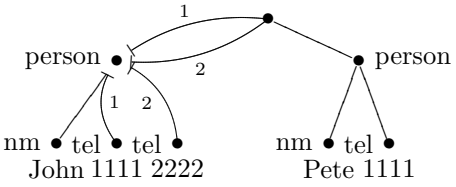

Figure 20: The first match is applied

would end up without a name in subtree *s2* and without a phone number in *s3*.

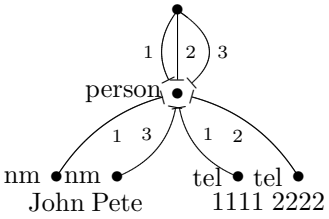

Figure 21: What would happen if the second match was also applied?

Of course, we could have chosen for an alternative solution in which the *param* value contains an array of numbers to which this subtree belongs, but then the document becomes less readable. The beauty of this solution is that a shared subtree doesn't need to be marked at all. Only when an inequality of subtrees occurs, we need to make such a distinction.

In general, a match must satisfy the following conditions: 1) if the original subtree contains a subtree that is replaced by a referencing node, then the matching subtree should only have equal nodes from there. It is possible to process those differences, like adding another referencing node to the original tree and give both referencing nodes their own *param* value. But then the complexity of the document increases enormously, 2) if the original tree has an ancestor with an *input* value and the node in the original tree is a shared node, then the corresponding node in the matching tree must be equal (see example), 3) if the original tree has an ancestor with an *input* value and the *param* value of the node in the original tree is equal to this *input* value, then the corresponding node in the matching subtree must not be the same, otherwise it will be treated as an unequal node and added to the original tree with its correct *param* value. But then we are adding equal nodes, which does not help to decrease the size of the document.

5.4.3 Applying Matches

If a match is valid, it can be applied. When a match is completely equal, it can be applied pretty straightforward. The matching subtree is replaced by a referencing node with a text node as a child containing the *id* number of the node to which is referred. If the original subtree has an ancestor with an *input* value, then this referencing node will also get this value as his *input* value. And finally for all the nodes that are replaced by the referencing node the new location will be set.

For the valid matches that contain differences, some extra steps are needed. In Listing 6 is shown in pseudo code what happens.

Listing 6: The pseudo code for the application of matches

```
applyMatch(origTree, nTree, refTree,
  inputOfOrigTree, inputOfNTree) {
  while(refTree != null) {
    if(refTree.type() == REFNODE) {
      //origTree is now a shared tree
      origTree.setShared();
      nTree.setNewLocation(origTree);
    } else {
      //a difference is detected
      clone = nTree.clone();
      origTree.getNewLocation().getParent().
        insert(clone);
      origTree.setParam(inputOfOrigTree);
      clone.setParam(inputOfNTree);
    }
    origTree = origTree.next();
    nTree = nTree.next();
    refTree = refTree.next();
  }
  //perform replacement
  nTree.setNewLocation(origTree);
  ref = new PxmlRefNode();
  text = new PxmlTextNode(origTree.
    getNewLocation().getID());
  ref.appendChild(text);
  nTree.getParent().replace(nTree, ref);
  ref.setInput(inputOfNTree);
  origTree.getNewLocation().
    setInput(inputOfOrigTree);
}
```

For the application of the match, we walk through the original tree, matching tree and *reftree* of the match in pre order. Whenever we encounter a referencing node in de *reftree* we mark the corresponding node in the original tree as being shared. We need this information when we check if a match is valid. If no referencing node is found, a difference between the original subtree and the matching subtree is detected. A clone of the complete subtree is now inserted after the node in the original tree. Both the original node and the cloned node will be getting a *param* value if they did not already have one. The *input* values of the trees are known on beforehand and passed through by the parameters of the method. If all differences are pasted into the original tree, the matching tree can be replaced by a referencing node. This is almost equal to the process mentioned above. But here, we also need to set the *input* values of the trees.

Notice that whenever values need to be set in the original tree, we always ask for the new location of the node in the original tree. For every node, the default value is the node itself. Only when the node is replaced, this variable will get a value other than itself. By asking for the new location of a node, we always get the correct information.

6 Experiments

To test whether compression techniques can benefit from the special properties of PXML, we set up some experiments. We will compare the results in order to draw conclusions. First, we will explain something about the measurement we chose, then the experimental environment will be discussed and finally the results will be shown.

6.1 Measurement

As mentioned before, we like to compare several techniques or combination of techniques in terms of compression ratio. In order to measure this, we need a formal definition of compression ratio. The definitions we found in the literature that are much used are: 1) the number of bits required to represent a byte, 2) the fraction of the input document eliminated. Since our prototype only compresses the DOM structure and is not yet incorporated in a XML database or application, we believe that measuring the size of the documents in terms of nodes is a better measurement. There are, for instance, databases that automatically use dictionaries for tag names or text fields and hence compress the document even further. To abstract from these implementation details, we choose for a measurement based on nodes.

First we specify exactly what nodes are. The number of nodes in a document (n_{total}) is the sum of the number of all elements ($n_{elements}$), all text nodes (n_{text}) and all attributes ($n_{attributes}$):

$$n_{total} = n_{elements} + n_{text} + n_{attributes}$$

The compression ratio r is defined as follows:

$$r = 1 - \frac{n_{total}^{after}}{n_{total}^{before}}$$

In this formula is n_{total}^{after} the number of nodes after applying the compression and n_{total}^{before} the number of nodes before compression.

Besides this measurement we are interested in the amount of overhead involved in the documents, due to the compression. Overhead nodes $n_{overhead}$ are the id, input and param attributes and the referencing nodes together, since these normally don't occur with these semantics in XML documents. The overhead o is then defined as:

$$o = \frac{n_{overhead}}{n_{total}}$$

6.2 Environment

We performed the experiments on a PC with the following specification: AMD Athlon 64 X2 Dual 3800+, 2,01 GHz, 2,00 GB RAM, Windows XP SP3, Java JRE 1.6.0-07

No further VM aruments were used. The experiments on one of the test documents failed, but failed as well with the largest heap space possible. This document is excluded from the results.

6.3 Data sets

For the experiments we need a number of XML documents with a different amount of uncertainty. These documents are then compressed by the different combinations of compression techniques.

This way we can see what influence the different techniques have, but also what the influence of the amount of uncertainty has.

The documents that we use come from the experiments done in [13]. This research investigated in the use of knowledge rules with data integration and fine tuning certain thresholdss to achieve better integration results. For the experiments they used data from the tvguide[1] and imdb[2] to integrate. They defined knowledge rules and varied the thresholds for these rules. The resulting documents contain a varying amount of uncertainty.

We use two sets of documents. It is beyond the scope of this paper to elaborate on all parameters that are involved in the generation of these documents. In short, in the first set of documents the extent to which is decided that two actors are the same, based on their name, is varied. And in the second set of documents the extent to which is decided that two movies are the same based on their title is varied. For each document in the set, the uncertainty is increasing.

Before we could actually use these documents we inserted the redundant probability and possibility nodes again, since this is required for the simplification method. This way, the size of the documents increased enormously since most of the redundant probability and possibility nodes were not generated during creation. However, this creates a fair start of the comparison after applying the different methods.

The size of the documents varies from 3177 nodes in the smallest document to 44815 nodes in the biggest document.

6.4 Results

We started the experiments by feeding the documents into the algorithm several times. Each document was compressed in each possible combination of compression techniques. In the diagrams we use abbreviations for the different methods: *SIMP* is the simplification method; *RRPP* the removal of redundant probability and possibility nodes; *PXML* is a combination of these two methods that are meant for PXML documents; *SIMP_DAG* stands for the combination of simplification and building a DAG. The rest is self-explanatory.

[1] http://www.tvguide.com
[2] http://www.imdb.com

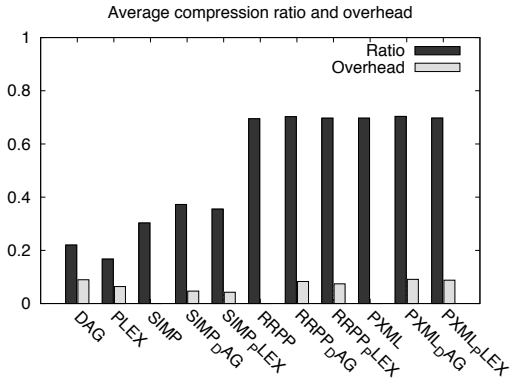

Figure 22: The average compression ratio and overhead for each compression method.

For each document, the results show a pattern that was quite similar for all the documents. So we took the average, which is depicted in Figure 22.

Notice that there is no overhead after applying SIMP, RRPP and PXML. Remarkable is that DAG scores better than PLEX on these documents. SIMP on its own, as well as combinations with DAG and PLEX scores better than DAG or PLEX solely, as expected. Same is true for RRPP and PXML. However, the ratio for combinations with RRPP or PXML are almost exactly the same. We will explore this later.

We can also see in Figure 22 that the amount of overhead decreases when we combine DAG or PLEX with the other methods. This is logical, since the other methods already achieve a certain amount of compression. The amount of nodes to which DAG or PLEX can be applied is then already decreased, hence the smaller overhead.

Let's take a more detailed look on the results. Figure 23 and Figure 24 show us the compression ratios for the first, respectively the second series. Because the first six documents of the first series resulted in almost the same values, we took the average value of the first and second set of three documents to keep the diagrams clear. Documents to the right at the x-axis have more uncertainty.

It is interesting to see that for both the series, DAG and for the first series, PLEX, benefit from

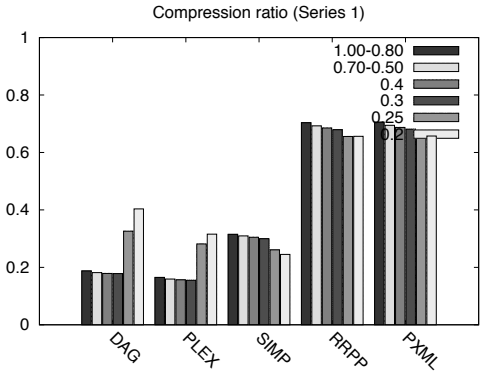

Figure 23: Compression ratio for the first series. Only singular compression modes.

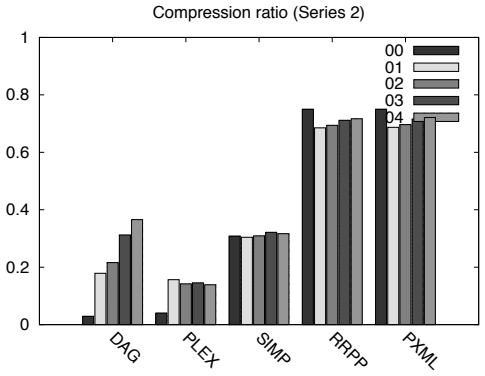

Figure 24: Compression ratio for the second series. Only singular compression modes.

the increasing amount of uncertainty. More uncertainty means more duplicated data and hence more chances for matches. Surprisingly, simplification did not benefit from the uncertainty. We expected that with increasing uncertainty more patterns (see 4.2) can occur in the document. But the number of conditions to satisfy for the second and third transformation in order to simplify is high. The first pattern will occur more often in documents with lower uncertainty instead. So combining this could be the reason why simplification does not benefit from uncertainty.

For RRPP it is not really a surprise that it doesn't benefit from uncertainty, because some probability and possibility nodes are now not redundant any more. And since PXML is the combination of SIMP and RRPP, this method doesn't benefit either. In the second series however, RRPP does benefit from uncertainty. This might be caused by the new (partially) duplicated subtrees that are added due to the uncertainty. If these subtrees are deep and don't have any useful probability and possibility nodes, then the number of redundant probability and possibility nodes increases, hence compression ratio actually increases.

We will now discuss the combination of the PXML methods with DAG and PLEX. In Figure 25 and Figure 26 we see, how these combinations perform with only simplification. Both diagrams show us that the combinations perform better when uncertainty increases. Especially the combination with DAG performs well. You might think that this is caused by the larger amount of overhead that naturally comes with PLEX, but the results do not confirm this. Another explanation for this, is that matches can be applied straightforward with DAGs, whereas with PLEX you need to check if previous matches did not change the subtree to which the match refers so that this match is not applicable any more.

For the combination of RRPP with DAG and PXML we see the same pattern, see Figure 27 and Figure 28. We already saw that for RRPP solely it could happen that it performed better or worse on documents with more uncertainty, dependent on the nature of the document. And here again we see that the combinations with DAG and PLEX benefit from the uncertainty, of which DAG benefits the most.

What is interesting in these diagrams is that even DAG and PLEX perform worse for the first few documents. For that, we don't have an explanation.

As we already could have seen in Figure 22 PXML has almost the same results as RRPP. It seems as if it doesn't matter whether or not the simplification is performed before RRPP. However, when we look at intermediary results, we see that SIMP significantly contributes to this compression ratio. It is just that, when you apply RRPP after SIMP you almost get the same end result as you would have without SIMP, see Table 1. It is important that we know that SIMP does in fact

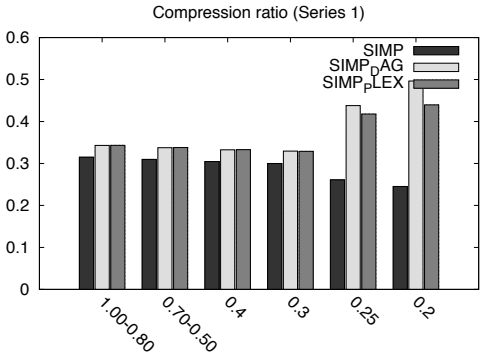

Figure 25: Compression ratio for the first series. All simplification variants.

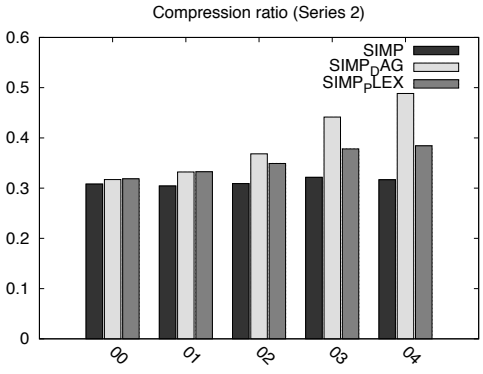

Figure 26: Compression ratio for the second series. All simplification variants.

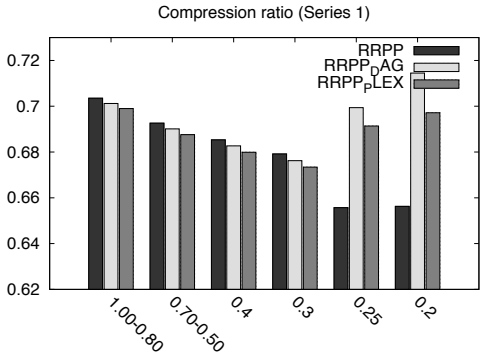

Figure 27: Compression ratio for the first series. All variants with removing redundant probability and possibility nodes.

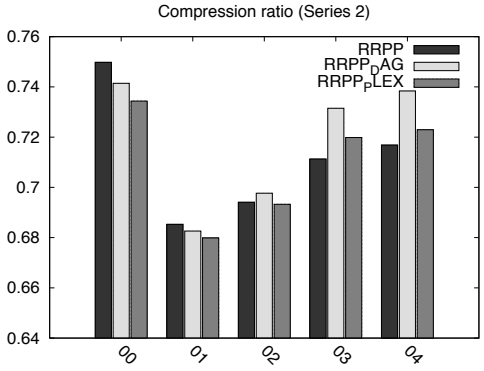

Figure 28: Compression ratio for the second series. All variants with removing redundant probability and possibility nodes.

contribute to the compression, which cannot be distracted from these diagrams.

From Figure 22 we could conclude that on average the methods that involve RRPP perform best. This is not a surprise. RRPP is the method that could and should always be used since it deletes the nodes you won't need and moreover no other method will be restricted by the removal of these nodes.

As the uncertainty increases in a document a combination of RRPP with DAG outperforms a combination with PLEX, see Figure 31 and 32. Although the differences between the results are small, we believe DAG is better than PLEX. Not only is the algorithm of PLEX more complex than DAG, it also results in documents that are more complex. This means that not only the compression itself could suffer from performance problems, also querying the document would take more time.

Although the results show us that combining with SIMP does not significantly affect the results, in certain situations we would recommend using a combination of SIMP though. SIMP changes the structure of the document. In case a human being

Table 1: Intermediary results for the first and last document of the first series.

Document	**1.00**		**0.20**	
n_{total}^{before}	9692		16041	
Mode	PXML	RRPP	PXML	RRPP
n_{total} after SIMP	6638	-	12111	-
n_{total} after RRPP	2822	2840	5496	5514
r (ratio)	0.709	0.707	0.758	0.757
Contribution of SIMP	52%	-	37%	-

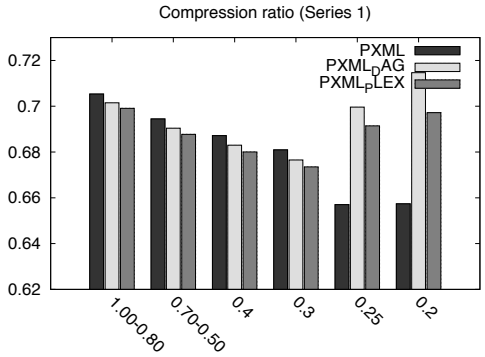

Figure 29: Compression ratio for the first series. All variants of both PXML methods.

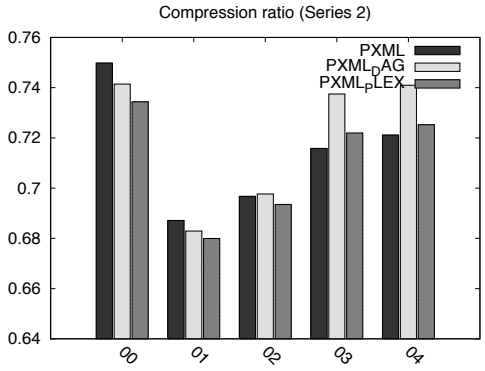

Figure 30: Compression ratio for the second series. All variants of both PXML methods.

needs to process the document, the simplification makes the document more convenient to read. It might also be the case that queries can be evaluated faster. We know that with simplification possibilities are pushed down. And in a certain way of query evaluation a search is made for the lowest common probability ancestor for certain XML nodes (Compare Paths Method, [12]). Intuitively, the distant to this probability node can become smaller by simplification.

7 Discussion

We have seen that the compression techniques can achieve high compression ratios, but it is remarkable that the PLEX method did not outperform DAG as in [2]. We know that the implementation of our PLEX method is not optimal. The algorithm is quite complex and hence we abstracted from some details. Take for instance the order of possibilities: in case of a movie named "Jaws" or "Jaws 2" in one possible world and "Jaws 2" or "Jaws" in another, we could probably merge them. In this prototype, we didn't.

Also, we used the DAG algorithm to base the algorithm for PLEX on. This caused matches to always include at least one leaf node. This restricted us in finding identical subgraphs. Fortunately, when information is duplicated due to uncertainty, this will often result in a two subtrees with one or more identical branches including the leaf. Still, when the algorithm is changed to accept matches without leaf node, we believe this could improve the performance of the PLEX method enormously.

Even if we succeed in improving the compression ratio of PLEX (but the same is true for DAG), is it worth it? The structure of the document is significantly changed. So it should be investigated if these documents can be queried easily. Is it possible to navigate through such a document conveniently? And what about the time it takes to compress, decompress, and query it? Will the algorithm be time and space efficient enough to handle larger documents? These are all questions that need to be answered in order to benefit from this method.

Another issue is the contribution of SIMP. In

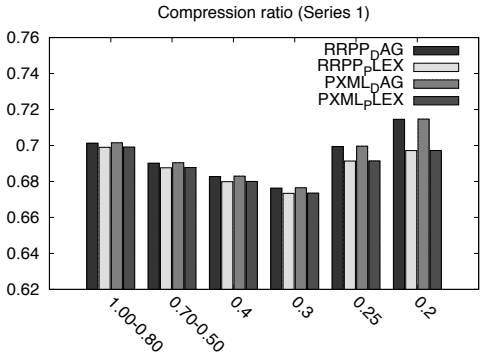

Figure 31: Compression ratio for the first series. Combinations of RRPP or PXML with DAG and PLEX.

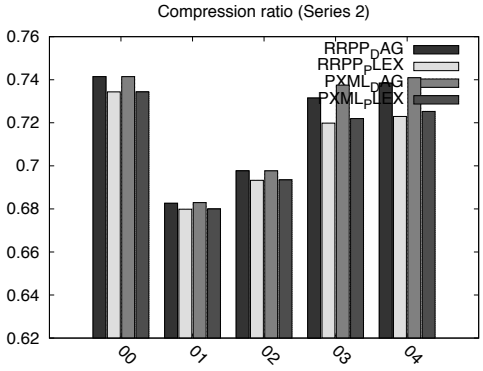

Figure 32: Compression ratio for the second series. Combinations of RRPP or PXML with DAG and PLEX.

terms of decrease of size, it might not be worthwhile to perform a complex method like this. However, the documents that are fed into the algorithms are not completely in PWS, since they would be inconveniently large. Thus the documents are already somewhat simplified. We could never do experiments with large documents in PWS since the prototype would not be able to handle them. But as we can see from Table 1 SIMP does contribute and hence changes the structure. As mentioned at the end of 6.4, we believe this structure could improve query evaluation, but research is needed here.

8 Conclusion

In this research we examined if we can improve the compression ratio of normal XML compression techniques by combining them with methods that are specially designed for PXML documents. From the literature we took the DAG and PLEX methods as existing XML compression methods and the simplification technique and the method in which redundant probability and possibility nodes are removed as the special PXML methods. We built a prototype in which these methods could be applied separately or combined.

With experiments we measured the compression ratio and the overhead for the different methods and combination of methods. We used documents with an increasing amount of uncertainty.

RRPP is the method that should always be used, since it removes useless nodes. The compression ratio can be improved by combining it with DAG or PLEX. Both methods show good increasing compression ratio by increasing amount of uncertainty, which is desired. DAG is preferred, since the algorithm and the resulting document are less complex than with PLEX.

In terms of size reduction, it is not worth the effort of simplifying the document, since RRPP alone achieves almost the same reduction. However, simplifying the document comes with a more simple structure of the document, that might increase performance when querying the document or even navigating. If we look only at the size of the document, performing SIMP is not worth it. Then the answer to the research question is: no, we cannot improve a standard XML compression technique by taking into account the special properties of PXML. If the compressed document needs to be queried, it needs further investigation.

Besides compression ratio we also measured the amount of overhead as a consequence of the DAG and PLEX method. The results gave no indication that the amount of overhead was problematic.

8.1 Future Research

Despite the promising results, the algorithms are not optimal yet. Thus the compression ratio could increase even more. But not only performance in terms of compression but also time and space complexity need to be researched and improved to be able to handle large documents.

Apart from the performance issue, it is important to see how these methods work in practice. In theory the compressed documents are still queriable, but research is needed here. And as mentioned before, we need to investigate if SIMP is worthwhile applying when a compressed document needs to be queried.

References

[1] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed xml. *Proceedings of the 29th VLDB Conference*, 2003.

[2] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of xml document trees. *Information Systems*, 33(4-5):456–474, 2008.

[3] J. Cheney. Compressing xml with multiplexed hierarchical ppm models. *Proceedings of the IEEE Data Compression Conference*, pages 163–172, 2000.

[4] M. Levene and P. T. Wood. Xml structure compression. *Proceedings of the Second International Workshop of Web Dynamics*, 2002.

[5] H. Liefke and D. Suciu. Xmill: An efficient compressor for xml data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 153–164, 2000.

[6] J.-K. Min, M.-J. Park, and C.-W. Chung. A compressor for effective archiving, retrieval, and updating of xml documents. *ACM Transactions on Internet Technology*, 6(3):223–258, 2006.

[7] W. Ng, W.-Y. Lam, and J. Cheng. Comparative analysis of xml compression technologies. *World Wide Web*, 9(1):5–33, 2006.

[8] W. Ng, W.-Y. Lam, P. T. Wood, and M. Levene. Xcq: A queriable xml compression system. *Knowledge and Information Systems*, 10(4):421–452, 2006.

[9] W. Ng, H. L. Lau, and A. Zhou. Divide, compress and conquer: Querying xml via partitioned path-based compressed data blocks. *World Wide Web*, 11(2):169–197, 2008.

[10] N. Sundaresan and R. Moussa. Algorithms and programming models for efficient representations of xml for internet applications. *Proceedings of the 10th International WWW conference*, pages 366–375, 2001.

[11] P.M. Tolani and J.R. Haritsma. Xgrind: A query-friendly xml compressor. *IEEE Proceedings of the 18th International Conference on Data Engineering*, 2002.

[12] R. van Kessel. Querying probabilistic xml. Master's thesis, University of Twente, April 2008.

[13] M. van Keulen and A. de Keijzer. Qualitative effects of knowledge rules in probabilistic data integration. Technical Report TR-CTIT-08-42, Centre for Telematics and Information Technology, University of Twente, Enschede, 2008.

[14] M. van Keulen, A. de Keijzer, and W. Alink. A possible world approach to uncertain relational data. *Proc. ICDE Conf.*, pages 459–470, 2005.