Compression of Probabilistic XML documents

Irma Veldman, Ander de Keijzer, and Maurice van Keulen

University of Twente
PO Box 217 Enschede
The Netherlands
{veldmani,keijzer,keulen}@cs.utwente.nl

Abstract. Database techniques to store, query and manipulate data that contains uncertainty receives increasing research interest. Such UDBMSs can be classified according to their underlying data model: relational, XML, or RDF. We focus on uncertain XML DBMS with as representative example the Probabilistic XML model (PXML) of [9]. The size of a PXML document is obviously a factor in performance. There are PXML-specific techniques to reduce the size, such as a *push down* mechanism, that produces equivalent but more compact PXML documents. It can only be applied, however, where possibilities are dependent. For normal XML documents there also exist several techniques for compressing a document. Since Probabilistic XML is (a special form of) normal XML, it might benefit from these methods even more. In this paper, we show that existing compression mechanisms can be combined with PXML-specific compression techniques. We also show that best compression rates are obtained with a combination of PXML-specific technique with a rather simple generic DAG-compression technique.

1 Introduction

Probabilistic XML (PXML) is XML that allows the representation of uncertainty in the data [9]. Uncertainty can, for example, arise from the integration of two or more XML documents when conflicts or ambiguities are encountered. Resolving these at integration time often is a severe obstacle, because it requires a huge amount of user effort. Being able to leave unresolved issues as uncertainty in the integrated document removes this obstacle, as they can resolved when they become visible during use, 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 [9]. For illustration purposes, we use the same running example as in [9]. The example concerns the integration 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 ambiguity and a possible 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.

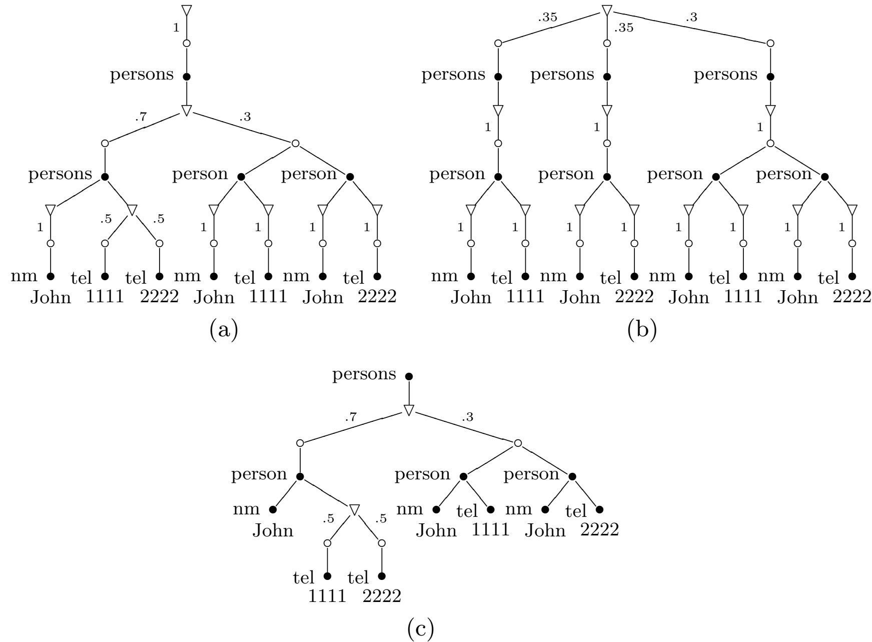

Fig. 1: Example probabilistic XML tree (a), its possible world style (PWS) tree (b), and its reduced form where probability and possibility nodes that give no extra information are omitted (c).

– 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. During integration these possible situations are represented as uncertainty in the PXML document. After integration, the probabilistic XML document could look like the PXML tree in Figure 1(a).

The $\triangledown$ nodes denote probability nodes. They represent choice points in the tree. Its children are possibility nodes that represent the mutually exclusive possible subtrees. The $\circ$ nodes denote the possibility nodes. They have an associated probability value. This value lies within the range $\langle 0.0, 1.0]$. The actual probability value is determined by a rule engine. The $\bullet$ nodes represent normal XML nodes.

Obviously it is important to be able to query the uncertain data represented by a PXML document. This is what [9] says about the semantics of 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."

Our example document represents 3 possible worlds (see Figure 1(b)).

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.

Efficient querying techniques do not suffer from this drawback, because they work directly on the compact representation [7]. 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 1(c) and section 3.

In this paper, we evaluate several combinations of PXML-specific and XML-generic compression techniques. We experimentally determine which combination has the highest compression ratio. We use a real-world data set originating from a probabilistic data integration application.

2 XML Compression

There are several ways to categorize compression mechanisms. In the comparative study of Ng etal. [4], 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 also group the compressors by their (in)ability of supporting queries. Due to space limitations, we only review queryable techniques here. For a complete review, we refer to [10]. PXML documents obtained from probabilistic data integration appear to contain many subtrees that are highly similar, but not completely equal. Therefore, we pay special attention to an advanced compression technique called BPLEX in Section 2.2, because it can compress parameterized subtrees.

2.1 Queriable compression techniques

XGrind [6] and XPress [3] 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).

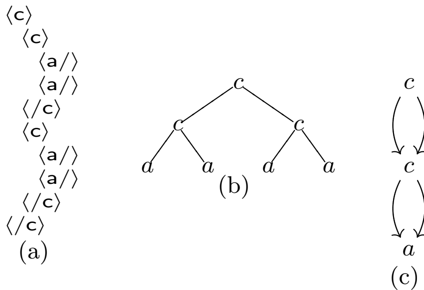

Fig. 2: Example XML fragment (a), its corresponding tree (b) and minimal DAG (c).

XPress [3] 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 [5] 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.

We illustrate DAG-compression with an example from [2]. It is based on the tree *c(c(a,a),c(a,a))*. Figure 2(a) shows the XML code of the example. The corresponding tree can be seen in 2(b). The minimal DAG for this tree is illustrated in Figure 2(c).

2.2 BPLEX

As mentioned, we pay some special attention to the compression technique, called BPLEX, of [2]. It takes the idea of transforming the XML tree into a DAG one step further. It is based on the sharing of common parameterized subgraphs

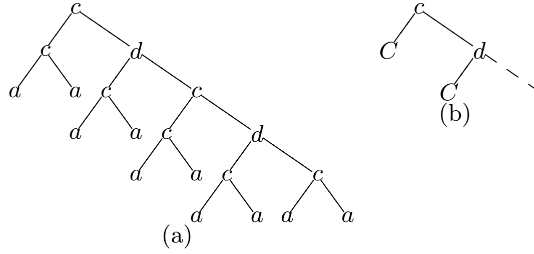

Fig. 3: Example XML tree (a) with the pattern $C \rightarrow$*c(A,A)* and $A \rightarrow a$ illustrated in (b) which shows the restrictions of DAG-compression.

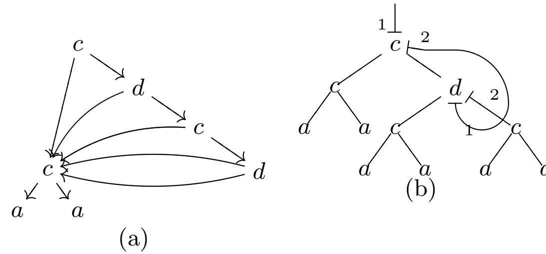

Fig. 4: The DAG created from Figure 3 (a) and the plexed version (b).

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. For example, the minimal DAG of Figure 2(c) 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 $W \rightarrow a$.

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 3(a). In this tree, there is a pattern (3(b)) that is repeated. Because different subtrees are hanging underneath, DAG-compression is not able to obtain sharing for this pattern. 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.

The sharing of the pattern is depicted in Figure 4(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.

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.

In Figure 4, we can also see the difference between creating a DAG from the tree in Figure 3 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].

3 PXML-specific Compression

[9] describes a PXML-specific method to compact PXML documents. This technique is different from generic XML compression techniques, because it produces a document that is fundamentally different according to XML semantics. The resulting document is, however, possible worlds equivalent, i.e., it represents the same possible worlds. We illustrate the push-down technique in Figure 5. We call this technique *simplification* in this paper. We refer to [10] for a more detailed description of the algorithm.

4 Experiments

To evaluate the effectiveness of the various combinations of compression techniques on PXML documents, we conducted some experiments. First, we briefly discuss the prototype implementation, and experimental setup and measurements.

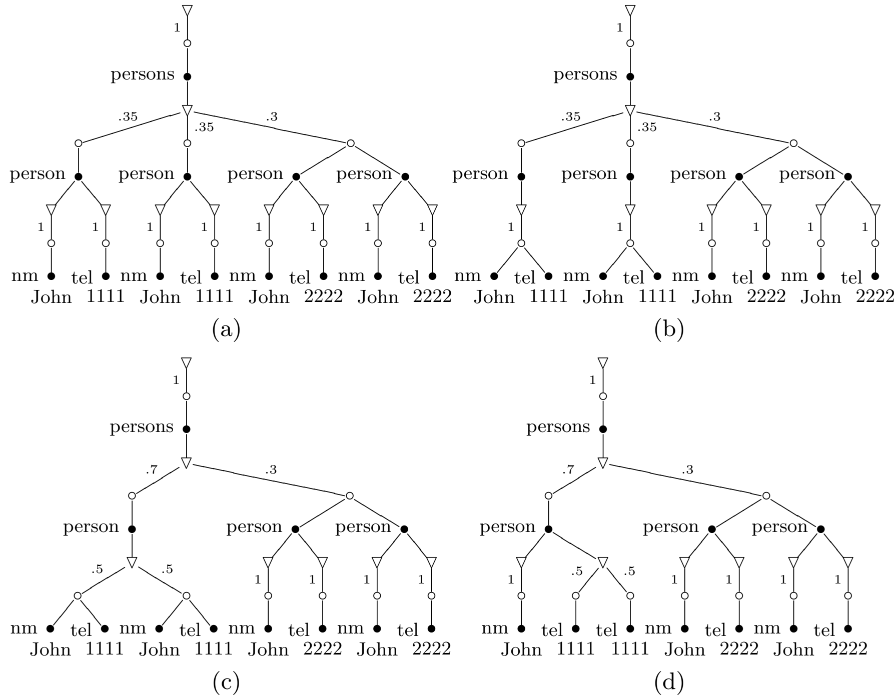

Fig. 5: Four example iterations of the PXML-specific push-down compression

4.1 Prototype

The main goal of the prototype is to measure compression ratios of PXML documents. The performance of the prototype in terms of compression and decompression speed will not be part of the comparison.

We adapted the BPLEX algorithm of [2] to work on a DOM structure of the tree instead of the SL grammar. We call this algorithm PLEX.

4.2 Measurement

We evaluate the compression techniques and combinations thereof based on *compression ratio*. Common definitions for compression ratio are: 1) the number of bits required to represent a byte, 2) the fraction in terms of bytes of the input document eliminated. Since there are many encoding techniques used to store XML in file systems or XML databases, we believe that a compression ratio measure based on a size in bytes is not suitable. XML databases, for instance, use indices for tag names and text fields, hence the byte size of a document is very system-specific. The compression techniques we focus on compress XML documents by reducing the number of nodes in the XML tree. We therefore measure compression ratios in terms of numbers of nodes.

The number of nodes in a document (n_{total}) is the total number of elements (n_{elements}), text nodes (n_{text}) and attributes ($n_{\text{attributes}}$): $n_{\text{total}} = n_{\text{elements}} + n_{\text{text}} + n_{\text{attributes}}$. The compression ratio r is defined as: $r = 1 - \frac{n_{\text{total}}^{\text{after}}}{n_{\text{total}}^{\text{before}}}$. In this formula, $n_{\text{total}}^{\text{after}}$ is the number of nodes after applying the compression, and $n_{\text{total}}^{\text{before}}$ the number of nodes before compression.

Besides this measurement, we are interested in the amount of *overhead* involved in compressed documents due to necessary additional bookkeeping. Typically, id's and other attributes need to be added to represent for example references. Overhead nodes n_{overhead} are all such attributes and nodes initially not present in the uncompressed documents. The overhead o is defined as: $o = \frac{n_{\text{overhead}}}{n_{\text{total}}}$.

4.3 Data sets

We use PXML documents representing the uncertain integration result produced by the probabilistic integration technique of [9]. To be able to investigate the influence of the amount of uncertainty in the document on the compression ratio, we use PXML documents obtained from integration under different conditions such as other integration rules and thresholds. The integration scenario used in [8] concerns integration of data from a TV guide [1] with data from IMDB [2].

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, each document is the result of integration based on one particular threshold. The threshold determines how similar two actor names need to be for the system decides that the two name possibly refer to one and the same actor. In the second set of documents, a different threshold is varied namely how similar two movie titles need to be for the system to decide that the titles possibly refer to the same movie. Varying both thresholds produces different amounts of uncertainty. Details can be found in [8]. The size of the documents varies from 3177 nodes in the smallest document to 44815 nodes in the biggest document.

4.4 Results

We feed the documents to various combinations of compression techniques. We use the following abbreviations for the different combinations: *SIMP* is the simplification method; *RRPP* the removal of redundant probability and possibility nodes; *PXML* is a combination of these two methods; *SIMP_DAG* stands for the combination of simplification and building a DAG. The rest is self-explanatory.

Figure 6 shows the average compression ratio and overhead over all documents in both sets for each combination of compression techniques.

Note that there is no overhead after applying SIMP, RRPP and PXML. Remarkable is that DAG scores better than PLEX on these documents. SIMP

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

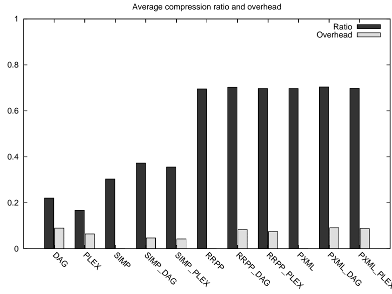

Fig. 6: Average compression ratio and overhead for each combination of compression techniques.

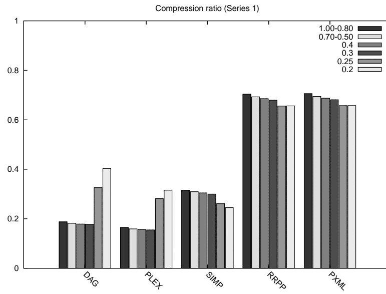

Fig. 7: Sensitivity of compression techniques to the amount of uncertainty (first document set)

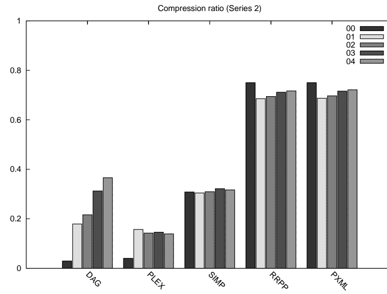

Fig. 8: Sensitivity of compression techniques to the amount of uncertainty (second document set)

on its own, as well as combinations with DAG and PLEX score 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 explore this later.

We can also see in Figure 6 that the amount of overhead decreases when we combine DAG or PLEX with the other methods. This makes sense, 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 7 and Figure 8 show the compression ratios for the first, respectively the second document set. Documents associated with thresholds that display highly similar compression ratios have been combined. The bars in the figures from left to right belong to documents with increasing uncertainty.

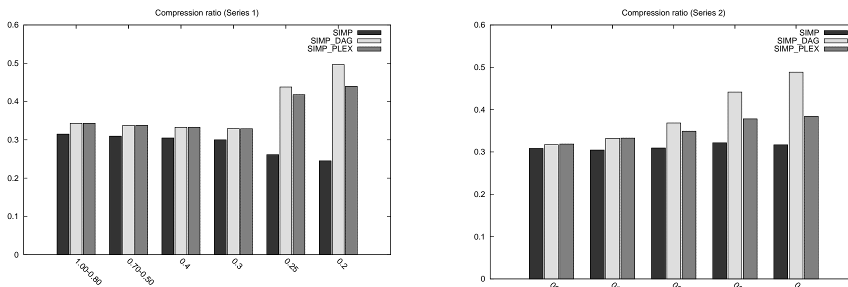

Fig. 9: Sensitivity of combinations with SIMP to the amount of uncertainty (first document set)

Fig. 10: Sensitivity of combinations with SIMP to the amount of uncertainty (second document set)

It is interesting to see that for both the series, DAG, and for the first series, PLEX, benefit from the increasing amount of uncertainty. More uncertainty means more duplicated data and hence more chances for matches. Surprisingly, simplification does not benefit from the uncertainty.

For RRPP it is not really a surprise that it doesn't benefit from uncertainty, because some probability and possibility nodes are not redundant any more. 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 increases.

We will now discuss the combination of the PXML methods with DAG and PLEX. In Figure 9 and Figure 10 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. One 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 one 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 11 and Figure 12. 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. Here again we see that the combinations with DAG and PLEX benefit from the uncertainty, of which DAG benefits the most.

As we have already seen in Figure 6, PXML has almost the same results as RRPP. It seems as if it doesn't matter whether or not the simplification is

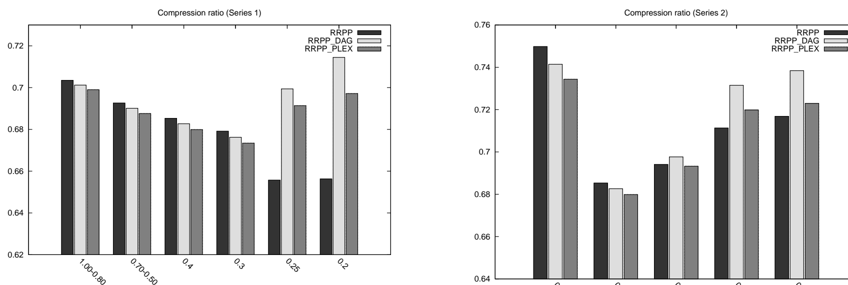

Fig. 11: Sensitivity of combinations with RRPP to the amount of uncertainty (first document set)

Fig. 12: Sensitivity of combinations with RRPP to the amount of uncertainty (second document set)

Table 1: Details 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%	-

performed before RRPP. However, when we look at intermediary results, we see that SIMP significantly contributes to the compression ratio. It is just that, when RRPP is applied after SIMP, one almost gets the same end result as without SIMP, see Table 1. It is important that we know that SIMP does in fact contribute to the compression, which cannot be distinguished in the diagrams.

From Figure 6, we can 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 redundant nodes, and no method is 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 15 and 16. 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.

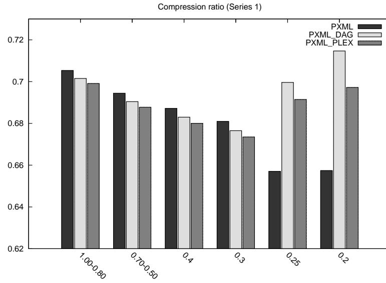

Fig. 13: Sensitivity of combinations with PXML to the amount of uncertainty (first document set)

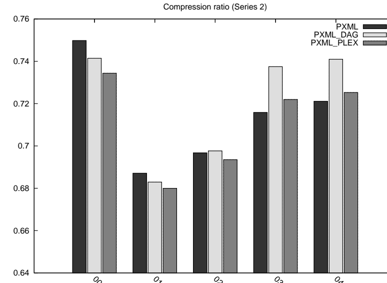

Fig. 14: Sensitivity of combinations with PXML to the amount of uncertainty (second document set)

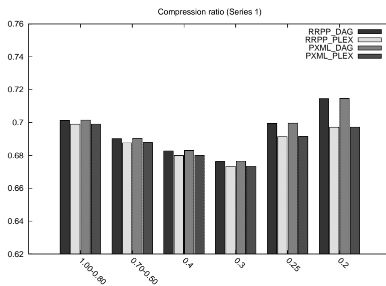

Fig. 15: Sensitivity of combinations with DAG and PLEX to the amount of uncertainty (first document set)

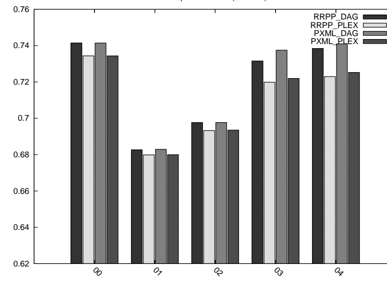

Fig. 16: Sensitivity of combinations with DAG and PLEX to the amount of uncertainty (second document set)

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 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 of which certain querying techniques can benifit. For example, the Compare Paths Method of [7] an often occurring operation is to determine the lowest common probability ancestor of certain XML nodes. Intuitively, the distance to this probability node can become smaller by simplification.

5 Conclusions

In this paper we evaluate the compression ratios of various combinations of compression techniques, both generic XML compression techniques as well as PXML-specific compression techniques. As representative techniques, we took the DAG and PLEX methods for generic XML compression techniques, and simplification and redundant nodes removal for PXML-specific compression techniques.

We experimentally evaluated 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 with increasing amount of uncertainty, which is desirable. 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.

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

Although, the compression ratio is substantial, it can be increased even more. Furthermore, for PXML documents to be used in practice, not only performance in terms of compression but also time and space complexity of the compression and decompression algorithms need to be improved to be able to handle large documents. Finally, document size is only one factor in query and update performance. It is an open problem how query algorithms can be adapted to support the compressed formats. Also the influence of these adaptations on query performance may change the suitability of the investigated combinations of compression techniques.

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. Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. A compressor for effective archiving, retrieval, and updating of xml documents. *ACM Trans. Internet Technol.*, 6(3):223–258, 2006.
4. W. Ng, W.-Y. Lam, and J. Cheng. Comparative analysis of xml compression technologies. *World Wide Web*, 9(1):5–33, 2006.
5. 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.
6. P.M. Tolani and J.R. Haritsma. Xgrind: A query-friendly xml compressor. *IEEE Proceedings of the 18th International Conference on Data Engineering*, 2002.
7. R. van Kessel. Querying probabilistic xml. Master's thesis, University of Twente, April 2008.
8. 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.
9. M. van Keulen, A. de Keijzer, and W. Alink. A possible world approach to uncertain relational data. *Proc. ICDE Conf.*, pages 459–470, 2005.
10. I.E. Veldman, A. de Keijzer, and M. van Keulen. Compression of probabilistic xml documents. Technical Report TR-CTIT-08-??, Enschede, May 2009.