Applying GRIPP to XML Documents containing XInclude and XLink Elements

Silke Trißl, Florian Zipser, and Ulf Leser

{trissl, zipser, leser}@informatik.hu-berlin.de

Abstract: XML documents have an inherent tree structure well suited for reflecting hierarchical relationships in data. However, data often also contain relationships, which are not hierarchical but, for instance, cyclic or many to many relationships. These relationships may be modeled by embedding XInclude or XLink elements into an XML document. Unfortunately, XML query languages usually cannot cope very well with these references. In this paper, we evaluate the applicability of GRIPP, a graph index-structure originally developed for indexing large biological networks, for enhancing XML query processing over XInclude and XLink elements. We show how such XML queries can be represented as graph queries and how they can be answered efficiently using GRIPP. Compared to XQuery formulations of such queries on top of the native XML database eXist, our approach reaches a more than 100-fold speed-up.

1 Introduction

XML today is the predominant format for modeling, storing, and exchanging hierarchical and / or semi-structured data. However, the nesting of XML elements resembles a tree, making the format less straight-forward to use for data that is not tree structured. Consider the DBLP[1] data set for example. DBLP uses XML to store information about publications. A publication contains a title, the authors, and the journal or conference the paper is published at. To store and query this information the hierarchical structure of XML is sufficient. But DBLP also provides cross-references to other publications. The W3C provides two possibilities to reference other parts of an document or other documents, namely XLink and XInclude elements. If we want to know if the publication with title "Extensible Markup Language (XML) 1.0"[2] is directly or indirectly referenced by the publication with the title "Accelerating XPath evaluation in any RDBMS."[3] we first have to look at the cross-references in the second paper and then recursively traverse the XML document to retrieve all cross-referenced publications. Clearly, to answer this query efficiently we should index the XML document before we start querying it. In this paper, we evaluate the applicability of the GRIPP index structure (GRaph Indexing based on Pre- and Postorder numbering), originally developed for indexing biological networks, to XML documents containing XInclude and XLink elements. We will index generated XML documents in

[1] See http://www.informatik.uni-trier.de/∼ley/db/

[2] by Tim Bray et al., Technical Report W3C.

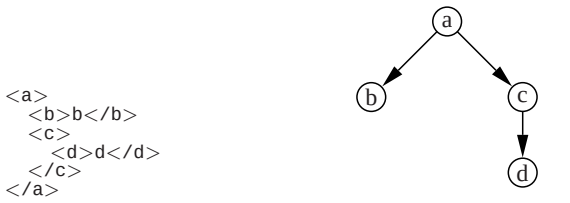
[3] by Torsten Grust et al.*ACM Trans. Database Syst*, 2004.

the size between one and 160 MB and varying number of XLink or XInclude elements. In addion we use the DBLP data set to show that GRIPP indeed is a very efficient option for certain types of queries on XML documents.

The reminder of the paper is organized as follows. In the next section we provide some background and related work. In Section 3 we describe GRIPP – our index structure and in Section 4 how to query GRIPP. Section 5 evaluates our approach compared to a native XML database and Section 6 concludes the paper.

2 Background

The eXtensible Markup Language (short XML) was developed by the W3C to store and exchange data. XML documents contain an implicit structure that is determined by the elements it contains. Every XML document contains a single root element and should be well-formed. An XML document is well formed if all elements of the document are closed only after all their child elements are closed. If we consider the elements as nodes and the implicit structure as edges every XML document forms a tree with a single root node. Figure 1(a) shows the XML document "a.xml", while Figure 1(b) shows the resulting tree.



(a) The XML document "a.xml". (b) The corresponding tree.

Figure 1: The XML document "a.xml" and the corresponding tree.

XInclude and XLink elements, which are necessary to express more complex relationships, destroy the inherent tree structure of an XML document. Both, XInclude and XLink elements address their target elements using XPath or XPointer expressions. But the semantic of the concepts is different, as specified by the W3C.

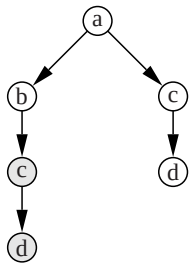
- **XInclude** [Vei06] allows to include other parts of an XML document, other documents or even other sources. When processing documents containing XInclude the included parts of the document are copied to the location, where the XInclude element is located.
- **XLink** [Orc01] just creates links either within or to other XML documents or to other sources with no fixed semantic.

This means, in the case of an XInclude element the specified target elements together with their successor elements are copied to the location of the XInclude element. Therefore an

XInclude element is only a placeholder for parts of a document as can be seen in Figure 2. Using XInclude elements, according to the W3C standard, the document can only form a directed, acyclic graph (DAG), i.e., a graph that contains no cycles.

```
<a xmlns:xi="http://www.w3.org/2001/XInclude/">
  <b>b
    <xi:include href=a.xml#c>
  </b>
  <c xml:id="c">
    <d>d</d>
  </c>
</a>
```

(a) The XML document "a.xml".



(b) The coresponding tree with expanded XInclude part (in gray).

Figure 2: The XML document "a.xml" containing XInclude elements and the corresponding tree.

In contrast XLink elements only point to elements in the same document or in other documents, but the content of the element is not copied. Therefore, XML documents containing XLink elements can express much more complex structures, including cycles. But at query time such XLink elements must be resolved. For a more detailed discussion on the use of pointers in XML documents see [Beh06].

Figure 3(a) shows an XML document containing XLinks. In Figure 3(b) the document is displayed as graph. Note, XLink elements are represented as dashed edges in the graph.

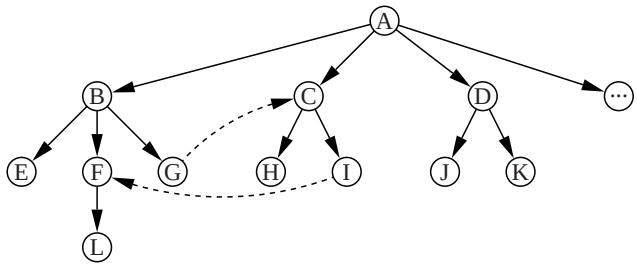
2.1 Related Work

For XML documents several strategies to index predecessor / successor axis exist. Grust et al. [Gru04] proposed to index XML documents using pre- and postorder values. Indexing XML documents using pre- and postorder values has the advantage that the order of child nodes is preserved, but has the disadvantage that it only works for trees. If you use this strategy for DAGs the index size grows exponentially as nodes with more than one incoming edge and their successors are labeled multiple times [Tri05]. Several groups addressed the problem to index and query XML documents containing XLinks. Schenkel et al. [Sch04] developed HOPI, a method based on the 2-Hop Cover. But creating the 2-Hop Cover requires n^3 time (n = number of nodes), which makes it inapplicable to large and highly interlinked XML documents. Chen and colleagues [Che05] labeled a spanning tree and stored non-tree edges in an separate index structure. They also proposed an algorithm to efficiently execute XPath queries. Their method works well for DAGs, but is not applicable to graphs. To address the indexing of graphs we developed the GRIPP index structure [Tri07] that is also based on the pre- and postorder labeling scheme. We will describe GRIPP in the following section.

```
<DBLP xml:id="A" xmlns:xlink="http://www.w3.org/1999/xlink/">
  <PAPER xml:id="B">
    <TITLE xml:id="E">Accelerating XPath Evaluation in any RDBMS</TITLE>
    <AUTHOR xml:id="F">Torsten Grust

      <ORGANIZATION xml:id="L">University of Konstanz<ORGANIZATION>
    </AUTHOR>
    <CROSSREF xml:id="G">
      <xlink:simple xlink:href="dblp.xml#C />
    </CROSSREF>
  </PAPER>
  <PAPER xml:id="C">
    <TITLE xml:id="H">Accelerating XPath location Steps</TITLE>
    <AUTHOR xml:id="I">
      <xlink:simple xlink:href="dblp.xml#F />
    </AUTHOR>
  </PAPER>
  <TECHNICAL_REPORT xml:id="D">
    <TITLE xml:id="J">Extensible Markup Language</TITLE>
    <AUTHOR xml:id="K">Tim Bray</AUTHOR>
  </TECHNICAL_REPORT>
  ...
</DBLP>
```

(a) XML document 'dblp.xml' containing XLinks



(b) Graph, G

Figure 3: XML document 'dblp.xml' and its graph representation G. Solid lines represent the inherent tree edges, dashed lines the edges introduced by XLink elements.

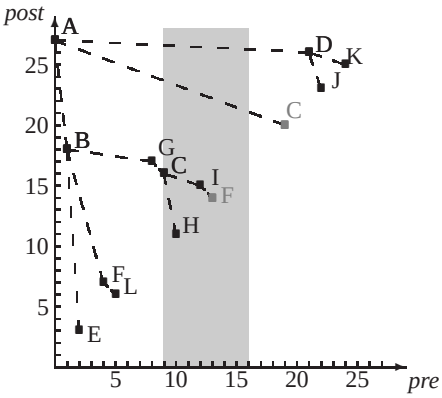
3 Indexing XML documents

We use GRIPP (GRaph Indexing based on Pre- and Postorder numbering) [Tri07] to index predecessor and successor relationships in XML documents containing XLink and XInclude elements. The basic idea of GRIPP is as follows. For now, we assume that the elements and relationships between elements of an XML document are stored as nodes and edges, i.e., as graph G in the database, which can be achieved easily when parsing the XML document. In GRIPP every node in G receives at least one pre- and postorder value during a depth-first traversal. The node together with the pre- and postorder value and the instance type (*tree* or *non-tree* instance) is stored as *instance* in the GRIPP index table, $IND(G)$. Clearly, in a graph a node v can be reached multiple times over different edges. During index creation, when v is traversed for the first time, we create a *tree instance* in $IND(G)$ and proceed the traversal. At any successive traversal of v we create a *non-tree instance* in $IND(G)$ and do not traverse any child nodes of v again. This means, in GRIPP

a node v has as many instances in $IND(G)$ as v has incoming edges in G. Therefore the size of the index $IND(G)$ is linear in the size of the indexed graph. The GRIPP index table $IND(G)$ for the XML document in Figure 3(a) is given in Figure 4(a).

node	pre	post	type
A	0	27	tree
B	1	18	tree
E	2	3	tree
F	4	7	tree
L	5	6	tree
G	8	17	tree
C	9	16	tree
H	10	11	tree
I	12	15	tree
F	13	14	non-tree
C	19	20	non-tree
D	21	26	tree
J	22	23	tree
K	24	25	tree

(a) Index table, $IND(G)$.



(b) $O(G)$, in gray $RIS(C)$

Figure 4: The GRIPP index table $IND(G)$ and the order tree, $O(G)$.

Using $IND(G)$ we can create the order tree, $O(G)$ as displayed in Figure 4(b). Every tree instance is an inner or leaf node, while non-tree instances are always leaf nodes in $O(G)$. The non-tree instance v'' of a node v has no child nodes in $O(G)$, but v has possibly child nodes in G. We therefore have to deploy a search strategy to answer reachability queries on XML documents using GRIPP.

4 Querying XML documents

We can use GRIPP to answer the XPath expression '`//v//w`' for arbitrarily shaped XML documents. To answer this XPath expression we use the GRIPP index table $IND(G)$. If you look at the resulting order tree $O(G)$ all reachable instances w of a node v must have a preorder value between the pre- and postorder value of v, i.e., $pre_v < pre_w < post_v$. This condition can be evaluated in an RDBMS with a single query.

But we face two problems when we use GRIPP. First, a node v may have many instances in $IND(G)$. But recall, every non-tree instance v'' of v is a leaf node in $O(G)$. We therefore always use the tree instance v' for querying. The second problem is that in the pre-/ postorder range of v' (also called *reachable instance set* of v, $RIS(v)$) we will only find instances of nodes that are reachable from v' in $O(G)$. We will miss nodes reachable from v in G, as during index creation we do not traverse child nodes when we insert a non-tree instance in $IND(G)$. To account for that we have to extend the search using the *hop technique*. To find all reachable nodes of v in G we basically perform a depth-first

search over the index using non-tree instances in reachable instance sets. In Figure 4(b) $RIS(C)$ contains a non-tree instance of node F, i.e., that node is a hop node for C.

To make the search more efficient we developed three pruning strategies, namely the simple, the skip, and the stop strategy. The simple and skip strategy avoid the repeated retrieval of a reachable instance set that has already been used during the execution of the query. For the stop strategy we have to precompute a list of stop nodes. If we reach a stop node s during the search we do not have to evaluate all non-tree instances in $RIS(s)$, which makes the search more efficient. For more details on the pruning strategies see [Tri07].

To answer '`//v//w`' we proceed as follows. We first find the tree instance v' of v and retrieve $RIS(v)$. If $w \in RIS(v)$ we finish and return `TRUE`, otherwise we have to use non-tree instances in $RIS(v)$ as hop nodes. We extend the search until we find an instance of w or no further usable hop nodes are available. Consider Figure 4(b) and '`//C//L`'. We first retrieve $RIS(C)$. We do not find an instance of L, but we find a non-tree instance of node F. We use F as hop node. In $RIS(F)$ we find an instance of L, therefore '`//C//L`' is `TRUE`. In [Tri07] we showed that this method is superior to other systems for many types of graphs, and especially for very large graphs.

5 Evaluation

We created XML documents of various sizes using ToXgene [Bar02]. All XML documents have a maximum depth of 5. As ToXgene can not create documents with XLinks or XIncludes we additionally inserted between 5 % and 20 % XLink and XInclude elements, respectively. We reference to other elements using XPointer with shorthand pointer. In our setting every XPointer points to exactly one XML element in a document. Before inserting an XInclude element we have to test, if the newly inserted XInclude element introduces a cycle, which would contradict the requirements of the W3C standard, and therefore should not be inserted. In addition, we used the DBLP dataset with a size of 345 MB containing 8.1 million elements and some 5,400 XLink elements.

To evaluate GRIPP applied to XML documents we compare it to the native XML database eXist [4]. For GRIPP we store the index table in a commercial database management system. We evaluate two different approaches to fill the GRIPP index table. In the first method we add pre- and postorder labels to elements while reading the document (called GRIPP*read*). For every non-XInclude or XLink element in the XML document we insert a tree instance to the GRIPP index table, while for every Xinclude or XLink element we insert a non-tree instance of the target element. In the second method (called GRIPP*db*) we create a graph corresponding to the structure of the XML document when reading the document. For the creation of the GRIPP index table we start the traversal at the node with the highest degree and traverse nodes according to their node degree as explained in [Tri07]. Both database systems are installed on a Dell dual Xeon machine with 4 GB RAM. For query times we compiled a set of 1,000 randomly chosen node pairs (for eXist only 100 due to memory restrictions) and averaged over query times.

[4] See http://exist.sourceforge.net/

5.1 Storing and indexing XML documents

Tables 1 and 2 show the time required to read, index, and store XML documents. For both, GRIPP*read*, where elements are labeled during reading, and eXist the time is dominated by reading the XML document. According to eXist the XML documents are also indexed during reading. GRIPP*db* requires more time due to the index creation inside the RDBMS. eXist can not read XML documents that are 60 MB or larger as a main memory exception occurs. We also did not test GRIPP*db* for huge documents (over 150 MB). The figures for documents containing XInclude elements are equivalent (data not shown).

Size	Nodes	Edges	GRIPP*read*	GRIPP*db*	eXist
1.2 MB	13,210	13,869	5.11	59.08	3.51
7.1 MB	64,693	67,926	17.46	674.80	69.78
13.2 MB	144,775	152,013	32.65	2,758.81	97.16
61.1 MB	644,240	676,451	157.38	55,489.54	–
152.4 MB	1,600,683	1,680,716	560.13	–	–

Table 1: Time in seconds to read, index, and store XML documents with increasing number of elements that contain 5 % XLink elements.

Size	Nodes	Edges	GRIPP*read*	GRIPP*db*	eXist
7.1 MB	64,693	67,926	17.46	674.80	69.78
7.3 MB	64,693	71,161	18.50	861.37	74.06
7.6 MB	64,693	77,630	19.88	1,057.69	94.98

Table 2: Time in seconds to read, index, and store XML documents with increasing number of XLink elements (5 %, 10 %, and 20 %).

5.2 Querying XML documents

To answer reachability queries we use user defined functions to query the GRIPP index in the database. For that we use the strategy described in Section 4. To query data stored in eXist we use XQuery.

Figure 5 show that the time required to query GRIPP*read* and GRIPP*db* is considerably faster than the time required to execute XQuery in eXist. For the document containing 13.2 MB GRIPP*read* requires 2.24 ms and GRIPP*db* 2.11 ms. In contrast, executing XQuery on eXist requires on average over 2,000 ms. Therefore, both GRIPP-based methods are for that size of the graph three orders of magnitude faster than eXist. In addition, the query times for eXist increases with increasing document size. In contrast, the query times of both GRIPP-based methods increases only slightly with increasing document size.

Figure 6 shows the average time required to query both GRIPP based approaches and to execute an XQuery on the eXist database for `//v//w`. The figures show that the query time for both GRIPP-based approaches as well as the times for eXist remain almost constant with increasing number of XLink elements.

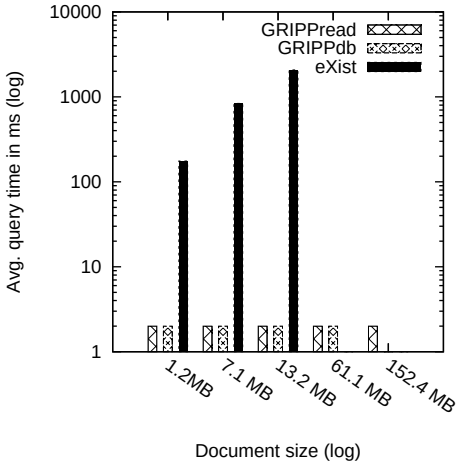


Figure 5: Average time in ms to answer `//v//w` on XML documents with increasing number of XLink elements.

For increasing document size as well as for increasing number of XLink elements querying GRIPP*db* is about 5 % faster than querying GRIPP*read*. The reason for this is, that the pruning strategies described in Section 4 are more efficient on the index created by GRIPP*db* than on that created by GRIPP*read*. The order tree $O(G)$ created by GRIPP*read* will only have a depth of 6, i.e., the depth of the XML document plus one level for the non-tree edges introduced by XLinks or XIncludes. In contrast during the creation of GRIPP*db* we will follow an XLink or XInclude when we reach the XInclude or XLink element for the first time and traverse all child nodes of the target node. Therefore the depth of the order tree in GRIPP*db* is much deeper. When querying the GRIPP*db* index with a node close to the root of such a long branch we cover more of the XML document immediately and can prune more hop nodes. The figures for XML documents containing XInclude elements are similar (data not shown). But the slight improvement of query times using GRIPP*db* compared to GRIPP`read` may not account for the big overhead in indexing time.

Indexing DBLP takes 21 minutes using GRIPP*read*. The average query time to answer reachability queries is 3.4 ms. With the size of 345 MB the DBLP XML document can not even be read into eXist, not to mention querying.

Concerning the comparison between GRIPP and eXist we have to note that executing XQuery `//v//w` on eXist does not result in `TRUE` or `FALSE`, but in the subtree of the query element and if XLinks were followed, in the subtrees of those as well. In contrast, GRIPP only returns `TRUE` or `FALSE`. But queries on GRIPP that have this feature of returning subtrees would take only marginally longer, as during querying we already search following nodes of query and hop nodes (the reachable instance sets).

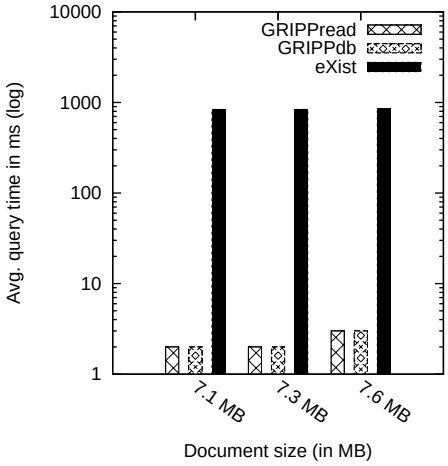


Figure 6: Average time in ms to answer `//v//w` on XML documents with increasing number of XLink elements.

6 Conclusion

This work presents a first step towards using graph indexes for XML queries with references. We showed how a graph index, such as GRIPP, can be used to index and query certain XPath axes in XML documents containing XLink and/or XInclude elements. In our experiments, such queries are evaluated up to three orders of magnitude faster than in the native XML database system eXist. However, there remain a bunch of open questions which we shall address in future work.

First, GRIPP is an index implemented in a relational database management system. Thus, integrating it into a native XML implementation such as eXist, Natix, or Tamino [5] is very difficult and probably not the right way to go. On the other hand, native XQuery implementations inside relational engines, such as the Viper engine in DB2 or the XQuery engine of Oracle, are still in an early stage and do not yet support user-defined extensions as they do for relational queries. Fortunately, a third class of XQuery systems translates XQueries into a sequence of normal SQL queries and execute those on top of a relational representation of XML data. Examples of such systems are Pathfinder [Rit07] or PPF [Geo07], both of which recently have been shown to outperform the other approaches despite the seeming overhead incurred by translating the XQuery into SQL. For those systems GRIPP is a perfect choice for indexing XML documents containing XInclude or XLink elements, since they anyway need a relational representation of the XML document. From this representation, the specific model expected by GRIPP capturing only the topological structure of the document can easily be extracted and, in a second step, indexed.

Other issues include updating of the index. Here techniques as described in [Wei05] may be used. Finally, for optimal performance a cost-based model for choosing when to use

[5] See http://www.softwareag.com/de/products/tamino/.

GRIPP and when to use a ”normal” traversal must be developed and integrated into an XQuery optimizer.

References

[Bar02] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons. ToXgene: a template-based data generator for XML. In *Proc. of the SIGMOD conference*, page 616, 2002. ACM.

[Beh06] E. Behrends, O. Fritzen, and W. May. Querying Along XLinks in XPath/XQuery: Situation, Applications, Perspectives. In *Proc. of the EDBT Workshop*, volume 4254 of *Lecture Notes in Computer Science*, pages 662–674, 2006. Springer

[Che05] L. Chen, A. Gupta, and M. E. Kurul. Stack-based Algorithms for Pattern Matching on DAGs. In *Proc. of the VLDB conference*, pages 493–504, 2005. ACM.

[Geo07] H. Georgiadis, V. Vassalos. XPath on Steroids: Exploiting Relational Engines for XPath Performance In *Proc. of the ACM SIGMOD conference*, pages 317–328, 2007. ACM.

[Gru04] T. Grust, M. van Keulen and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29:91–131, 2004.

[Orc01] D. Orchard, S. DeRose, and E. Maler. XML Linking Language (XLink). *Technical report*, W3C, June 2001.

[Rit07] J. Rittinger, J. Teubner, and T. Grust. Pathfinder: A Relational Query Optimizer Explores XQuery Terrain. *BTW*, pages 617–620, 2007.

[Sch04] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An Efficient Connection Index for Complex XML Document Collections. In *Proc. of the EDBT conference*, volume 2992 of *Lecture Notes in Computer Science*, pages 237–255, 2004. Springer.

[Tri05] S. Trißl and U. Leser. Querying Ontologies in Relational Database Systems. In *Proc. of the DILS workshop*, volume 3615 of *Lecture Notes in Computer Science*, pages 63–79, 2005. Springer.

[Tri07] S. Trißl and U. Leser. Fast and Practical Indexing and Querying of Very Large Graphs. In *Proc. of the ACM SIGMOD conference*, pages 845–856, 2007. ACM.

[Vei06] D. Veillard, J. Marsh, and D. Orchard. XML Inclusions (XInclude). *Technical report*, W3C, Nov 2006.

[Wei05] F. Weigel, K. U. Schulz, and H. Meuss. The BIRD Numbering Scheme for XML and Tree Databases - Deciding and Reconstructing Tree Relations Using Efficient Arithmetic Operations. In *Proc. of the XSym conference*, pages 49–67, 2005.