XQuery Optimization

Philippe.Michiels@ua.ac.be

University of Antwerp, Belgium

Abstract

We take a closer look at the optimization problems that are associated with the XQuery language. We discuss the research that has been done and some open problems along with potential solutions. XQuery is a quite young standard and many issues with respect to optimization remain unresolved. This work will focus on three aspects of optimization in XML databases being: duplicate removal, XML algebras and indices. It is done in close cooperation with the development team of the Galax XQuery Engine from AT&T Labs and Lucent - Bell Labs. Eventually, we aim to implement our optimization results into the Galax engine.

1 Introduction

Over the past five years, XML has gained great popularity as a universal data exchange format. As it grows towards becoming the standard for e-business, the amount of exchanged XML-data grows exponentially. Along with this, the requirement to store and query XML efficiently increases rapidly. In this context, many query languages for XML such as Lorel, Quilt, XQL, XML-QL, XPath and XQuery have appeared [4].

The two most popular among these query languages are XPath and its superset, XQuery.
XPath/XQuery is being developed by the XML Query working group at the World Wide Web Consortium (W3C) in an effort to provide users with a powerful means to query and transform XML. There are two important groups that have influenced the development of XPath/XQuery.

- The first group, the *document community* is mainly interested in XML as a humanly consumable exchange format. They are continuously looking for the development of new tools that allow the easy manipulation and transformation of relatively small XML documents.
- The second group, the *data community* is mainly interested in XML as a data storage format. Their primary focus is the design of query languages and storage methods to select data from vast amounts of XML data efficiently.

Until quite recently, there wasn't much interest in the optimization problems posed by XPath and XQuery. The reason for this was that XML was primarily used as a human consumable exchange format rather than a large scale data storage format. The optimization of queries on small documents seems not very useful. But as the usage of XML shifts towards the data-oriented paradigm, more efforts are done to allow the efficient evaluation of XPath/XQuery.

2 Existing Work

XQuery is a strongly typed functional language that supports both the transformation and querying of XML documents. It allows the extraction of data through XPath expressions, the joining of several documents, and the construction of completely new documents. Despite being a strongly evolving language, there are various implementations around these days. Depending on the context of every implementation (document oriented vs. data oriented) various implementation and optimization techniques are used and there is active debate about which techniques are the best for an XQuery implementation.

XPath Complexity The bulk of the research in optimizing query languages for XML has been done in XPath. This seems evident since XPath is a simpler, older and more stable subset of XQuery. In observing existing implementations of XPath, there is clear evidence that most of them are inspired by the document community as their execution times show exponential behavior. However, it is possible to compute the result of an XPath query in polynomial time and fragments of the language exist where evaluation can even be done in linear time [7]. Research has shown that XPath query evaluation is P-hard [8, 18].

Pipelined evaluation Other research projects focus on the parallel execution possibilities of XPath. One approach would be to translate XPath into algebraic expressions parameterized by programs that are mainly built to perform navigational primitives like accessing the first child, etc. This approach allows the pipelined evaluation of the queries. Therefore, the generation of duplicate nodes must be avoided because duplicate removal is a pipeline breaker [11]. As an illustration of the problem we take a look at the example from [11]. Because every location step results in an ordered set of nodes, a simple `UnnestMap` operation is an appropriate algebraic operator for representing it.

$$\texttt{UnnestMap}_{\$i:l}(e) := \{x \circ [\$i : y] | \, x \in e, y \in x/l\}$$

where $\$i$ is an attribute name of the table in which results are stored, l is a step expression, e a general XPath expression and $\circ$ denotes tuple concatenation. This operation does not eliminate duplicates, it uses bag semantics. If we look then at the translation of the path expression `//A//A` into `UnnestMap` operations:

$$\texttt{UnnestMap}_{\$2:\texttt{desc}::A}(\texttt{UnnestMap}_{\$1:\texttt{desc}::A}(.)),$$

it is clear that if this expression is applied to the root of an XML document that is a binary tree of only `A` elements, the result will contain duplicates. This is a common problem with straightforward translation of XPath expressions. Removing duplicates after every `UnnestMap` operation avoids unnecessary work but jeopardizes pipelining. Also, inserting unnecessary duplicate removal operations may also have a bad influence on the evaluation time. One possible solution to this problem is to avoid duplicate generation in the first place. This can be achieved by applying rewriting rules to duplicate generating XPath expressions and translating them into step functions [17, 11].

Schema-Based Optimizations Many XML-documents follow a schema. We can use this schema-knowledge to perform some optimizations on XPath expressions by trying to eliminate impossible path expressions (i.e. expressions that are know to be always empty) or to remove redundant conditions, etc. For instance, when looking at the following DTD excerpt:

```
<!ELEMENT Students (Student*)>
<!ELEMENT Student (Name, Address, Birthday)>
<!ELEMENT Address (Street, City, Zip,
                             (Tel|Email))>
```

and if we would like to query the document for all students that provided their birthday and phone or email contact, then our query would look as follows:

```
//Student[Birthday]/Address[Tel|Email]
```

According to the DTD, however, $Birthday$ is a required child element of $Student$ and an $Address$ always requires a Tel or $Email$ child element. So the query is equivalent to

```
//Student/Address
```

To make such optimizations possible, [15] proposes to compute so-called *path equivalence classes* (PEC) from DTDs. PECs represent path dependencies that hold for any XML document that conforms to the DTD. Path dependencies basically are statements about admissible occurrences of pairs of paths with regard to a DTD. At query compile time, this information is used to (a) remove redundant conditions, (b) simplify conditions, and (c) detect contradictory conditions and satisfyability.

Formal Semantics Another approach for optimization of both XPath and XQuery is to start from their core mapping, defined by the formal semantics [2] . This translation into the functional language that the XQuery Core is, causes both XPath and XQuery to benefit from the many optimizations that exist for functional programming languages and for other database languages [1].
We illustrate the formal semantics' core mapping with an expression that queries the example document in fig. 1:

```
<?xml version ="1.0"?>
<PersonList Type="Student">
  <Title Value="Student List"/>
  <Contents>
    <Person>
      <Name>John Doe</Name>
      <Id>111111111</Id>
      <Address>
        <Number>123</Number>
        <Street>Main St</Street>
      </Address>
    </Person>
    <Person>
      <Name>Joe Public</Name>
      <Id>666666666</Id>
      <Address>
        <Number>666</Number>
        <Street>Hollow Rd</Street>
      </Address>
    </Person>
  </Contents>
</PersonList>
```

Figure 1: An example XML document.

```
//Person[Address/Street = "Hollow Rd"]
```

After normalization [2], the query (roughly) looks as given in figure 2.

```
ddo(
  for $glx:dot in ddo(document("example1.xml"))
  return
    ddo(for $glx:dot in
        ddo(desc-or-self::node()) return
          ddo(for $glx:dot in ddo(child::Person)
          return
           if (some $v3
            in fn:data(
              ddo(for $glx:dot
                  in ddo(child::Address)
                  return child::Street))
            satisfies fn:boolean(
              some $v5
              in fn:data("Hollow Rd")
              satisfies fn:boolean(
                op:equal(
                  cast as xsd:string (data($v3)),
                  cast as xsd:string ($v5))
              )))
       then ($glx:dot)
       else ()
)))
;
```

Figure 2: Example core mapping

The `ddo` function calls are the `distinct-docorder` operations used in the formal semantics to guarantee that (intermediate) results are in document order and contain no duplicate nodes.
We see that there is not much left of the original XPath expression, which is torn apart in several `for` loops iterating over the result of the several axes.
In the course of the standardization process of XQuery, a reference implementation called Galax [3] is being developed. Galax features a portable lightweight XQuery implementation which has the great advantage of being strongly

related to the XQuery Formal Semantics [2] which provides us with a solid base for researching possible optimization techniques [1].

3 Open Problems

Duplicate elimination We earlier discussed the pipelining problem where the eventual goal was to *prevent* the generation of duplicates, because their elimination breaks the pipeline. However, it is not always possible to rewrite every expression so that no duplicates can occur [11]. Since the semantics require the result of any XPath expression to be in document order and free from duplicates, we will always have to provide some kind of duplicate elimination.
The XQuery Formal Semantics try to enforce this property by providing an explicit mapping to the XQuery core language, inserting *distinct-doc-order* meta-instructions (see example core mapping). By mapping queries to their core representation, we do ensure correctness. But a new problem arises: the elimination of duplicates after every intermediate result can be very time consuming, which causes performance problems. In Galax [3], it even has proved to be one of the major bottlenecks.
Our current research focuses on finding an algorithm or a set of rules, that can be used to find out whether or not the elimination of duplicates at some point in the query is necessary. For instance, the query

```
/child::*/descendant::*
```

is mapped to the core as follows:

```
ddo(for $glx:dot in ddo(input()) return
      ddo (for $glx:dot in ddo(child::*) return
       descendant::* ))
```

Within Galax, this is equivalent to the expression without any of the `ddo` meta instructions since the implementation of its axes does not generate duplicates for this query, no matter on what document it is used.
In a first attempt – and to get a clear understanding of the problem – we are looking at a subset of the XPath language, hoping to find a complete set of rules for avoiding the unnecessary elimination of duplicates. Naive approaches that only take the properties *order* and *duplicate freeness* into account have failed at the completeness level. For instance, the following inference rule for *order* causes a problem for completeness:

$$\frac{a \in \{\texttt{self}, \texttt{child}\} \wedge path \vdash order}{path/a :: nt \vdash order}$$

There are path expressions whose result is always ordered, even when they are normalized without inserting explicit sorting and duplicate removal operations. This property however, is not derived by this rule. The following path expression always returns an empty result and therefore this result is always ordered:

```
/child::*/parent::a/child::*/parent::b
```

But the rule given above does not derive this property. We will have to take additional properties into account. Possible candidates are:

- *emptyness* - indicating that the result of the subquery will always be empty;
- *singleton (at-most-one)* - indicating that the result of the subquery will always contain exactly (or at most) one element and therefore is always sorted and duplicate-free;
- *unrelatedness* - indicating that the result of a subquery contains related nodes (i.e. nodes that have an ancestor-descendant relationship);
- *generation* - indicating that the result of a subexpression contains only nodes that are of the same generation (i.e. nodes that are at the same level in the tree hierarchy).

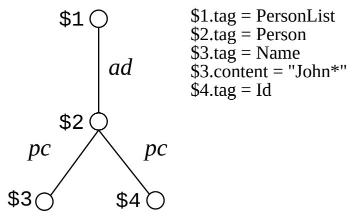
After studying this XPath subset, we will move on to extend our approach to the full XQuery syntax. The algorithm will eventually be implemented in the Galax XQuery engine.

An XQuery algebra Just as there are many query languages for XML, there are also quite a lot of algebras for it. Defining the right algebra is feeding an ongoing debate in the XML Query community that started a few years ago. However the formal semantics[1] provide a good base for optimization, there are some disadvantages in that approach too. Mapping XQuery to its core representation makes it harder to perform some optimizations. For instance, determining the need for removing duplicates or sorting the result inside XPath expressions may prove to be much easier from the surface syntax then from the core mapping. On the other hand, some typical database optimization techniques like join reordering are easier to achieve when we have an algebra that has explicit (join-)operators. During the development of Galax, the need for a fully fledged algebra became clear as the limits of the XQuery Core got closer.
There are many different approaches in designing an algebra for XML query languages. One approach, which is discussed in XAL [5], defines a set of logical operations closely matching the XQuery operations. The query is then represented as a tree, on which algebraic operations are performed to optimize the query plan. Since the XQuery Core expressions can be translated to an abstract syntax tree that closely matches this algebra (as is done in Galax), this approach could be usefull in query engines based on the formal semantics.
On the other hand, it is well accepted that a set-oriented algebra is essential for effective query processing in a database system. This is why TAX (Tree Algebra for XML)[12, 5] proposes a natural extension of the relational algebra, with a small set of operators. The result is a logical tree algebra, which uses *tree patterns* to represent queries. For example, the following query selects all persons whoes name starts with 'John' and that have an Id child element.

[1]There is discussion about the status of the formal semantics as an algebra.

```
//Person[name="John*" ^ Id]
```

This query's tree pattern looks as follows:



The tree pattern's edges are labeled with the relationship between the two nodes. This can be *pc* for a parent-child relationship or *ad* for an ancestor-descendant relation. The corresponding algebraic expression looks as follows:

$$\pi_{\mathcal{P},PL}(\mathcal{C}),$$

where $\mathcal{P}$ is a pattern tree, $\mathcal{C}$ is the input collection and PL is the projection list, which is a list of node labels appearing in the pattern $\mathcal{P}$ (ie {`$1, $2`}). As in the relational algebra, identities are used to consider alternative execution plans. However, while this algebra is very powerful with respect to optimization possibilities, it does not cover the full expressive power of XQuery (yet).
Finding the ideal algebra for XML has already proven to be very hard. It is clear that the algebra should take XML-order into account and be powerful enough to cover the full expressive power of XQuery. Moreover, it should enable us to use a broad variety of optimization possibilities in order to find better query plans. Most current algebras have problems with one of these requirements.

Indexing XML Another well-known way to improve query execution time is to build indices. A great effort has already been done in the research for indexing semi-structured data and XML. Various indexing schemes can be found in specialized literature [6, 16, 14, 13, 9, 10] . We discuss a few of them.
The most simple kind of index is the *name index*. Name indices use tag names to link a list of matching nodes to a corresponding index entry. This index has more advantages than its simplicity alone. It is easy to maintain and because one can choose which element names are indexed, there is always a possibility to control the size of the index. There are, of course, disadvantages too. For instance, in XML a tag can occur as a subelement of two different parent tags, so additional efforts will have to be done if only the subtags of one of these two parent tags are requested. Very similar to this index, is the value index, which uses the values inside elements as an indexing key.
More advanced techniques use the document structure for building indices. This structure describes the XML document by a so called 'minimal schema', which is nothing more than a list of allowable node interconnections. There are few different kinds of *structural indices*:

- **Dataguides** [6] are a short description for semi-structured databases. They describe every labelpath of a document exactly once and there is no path in a dataguide that is not in the XML database. The problem with dataguides is that they lack the notion of document order, which rules out a wide range of queries.

- **Template Indices** [16] are used to accelerate the evaluation of path expressions that match a certain template. A secondary goal of template indices is to reduce the size of the index by making sure that every node in the indexed document appears only once in the index, which is not always the case with dataguides. In general, template indices use backward bisimulation to arrange document nodes into equivalence classes. Nodes in the same equivalence class are reachable through the same template path. A disadvantage of such an index is that it can get very large.

- **A(k)-indices** [14] Indexing very long paths that are rarely accessed, makes the index unnecessarily big. A(k)-indices avoid these kinds of problems by indexing only paths that have a length which is less than k. An approximation is used for answering longer queries.

- **Forward & Backward indices** [13] are built to cover all possible queries that fall inside a subset of XPath, called branching path queries. The biggest problem with these kinds of indices is their size. A solution for this is to consider infrequently accessed tags as having the same label, giving priority to tree-edges over *idref* edges or exploiting local similarity as in A(k)-indices.

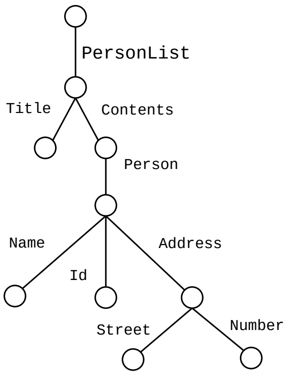


Figure 3: A minimal schema can be used to construct structural indices.

Most of the above indexing approaches focus on providing efficient support in the evaluation of sequences of `child`, `descendant` and `descendant-or-self` axes, which denotes an extremely limited subset of XPath. This is why [9] proposes an entirely different database index structure for XML, which supports *all* axes evaluated from *arbitrary* context nodes. The latter establishes support for efficient evaluation of XPath queries that are embedded in XQuery expressions. The index can be implemented using only relational techniques but it takes advantage of R-tree support in some database systems.

The problem with most of the indices is that upon an update, the index has to be reconstructed partly or entirely. With the exception of dataguides, most of the indexing techniques are not really practical due to the cost of index maintenance. Many of the indexing systems discussed here, do not take order into account because they were originally designed to operate in semi-structured environments. Interesting research topics in this area are the determination of optimal indexing schemes, the search for efficient algorithms to keep indices up-to-date and the comparison of different indexing schemes.

4 Conclusion

Solving the duplicate removal problem is a hard nut to crack. There are two goals that are somewhat contradictory: we want to avoid expensive duplicate removal or sorting operations as much as possible but by doing this we allow the amount of duplicates in the intermediate results of an XPath/XQuery expression to grow exponentially. Currently the Galax engine implements sorting and duplicate removal after every step expression in an XPath subquery, which results in a bottleneck upon querying larger documents. A selective elimination of a large part of these `distinct-docorder` operations is a serious optimization. We are currently working on this problem.
Currently Galax is using a variant of the XQuery core with support for tuples as an algebra. Using an algebra that is more adequate for certain optimizations is desirable, but what is the right algebra for XML Query processing is still largely an open issue. There are a couple of good candidates but most of them have a few important shortcomings such as not counting in the fact that XML is an ordered data structure or being not powerful enough to cover the full expressive power of XPath/XQuery.
Indices are an efficient way to speed up query evaluation in database systems. There are various indexing systems for use with XPath, but only a few of those seem very useful in XQuery. Adjusting XPath indexing schemes for XQuery and extending them is an interesting research topic.

References

[1] B. Choi, M. Fernández, and J. Siméon. The XQuery formal semantics: A foundation for implementation and optimization. `http://www.cis.upenn.edu/~kkchoi/galax.pdf`, 2002.

[2] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, 2002. `http://www.w3.org/TR/query-semantics`.

[3] M. Fernández and J. Siméon. *Galax, the XQuery implementation for discriminating hackers*. AT&T Bell Labs and Lucent Technologies, v0.3 edition, 2003. `http://www-db-out.bell-labs.com/galax`.

[4] M. Fernández, J. Siméon, P. Wadler, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, and J. Widom. XML query languages: Experiences and exemplars, 1999.

[5] F. Frasincar, G.-J. Houben, and C. Pau. XAL: An algebra for XML query optimization. In *ADC 2002*, Melbourne, Australia, 2002. ACS.

[6] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB 1997*, pages 436–445. Morgan Kaufmann, 1997.

[7] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB 2002*, Hong Kong, 2002.

[8] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *ACM SIGMOD (PODS) 2003*, San Diego (CA), 2003.

[9] T. Grust. Accelerating XPath location steps. In *ACM SIGMOD 2002*, pages 109–120, Madison, 2002.

[10] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *VLDB 2003*, 2002.

[11] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In *WISE 2002*, pages 215–224, Singapore, 2002.

[12] H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. Tax: A tree algebra for XML. In *DBPL 2001.*, 2001.

[13] R. Kaushik, J. F. Naughton, P. Bohannon, and H. F. Korth. Covering indexes for branching path queries. In *ACM SIGMOD 2002*, pages 133–144, 2002.

[14] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *ICDE*, 2002.

[15] A. Kwong and M. Gertz. Schema-based optimization of XPath expressions. Technical report, Univ. of California, dept. of Computer Science, 2001.

[16] T. Milo and D. Suciu. Index structures for path expressions. *Lecture Notes in Computer Science*, 1540:277–295, 1999.

[17] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *XMLDM 2002*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

[18] L. Segoufin. Typing and querying XML documents: Some complexity bounds. In *ACM SIGMOD (PODS) 2003*, San Diego (CA), 2003.