Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions

Mary Fernández[1], Jan Hidders[2], Philippe Michiels[2*], Jérôme Siméon[3**], and Roel Vercammen[2*]

[1] AT&T Labs Research
[2] University of Antwerp
[3] IBM T.J. Watson Research Center

Abstract. XQuery expressions can manipulate two kinds of order: *document order* and *sequence order*. While the user can impose or observe the order of items within a sequence, the results of path expressions must always be returned in document order. Correctness can be obtained by inserting explicit (and expensive) operations to sort and remove duplicates after each XPath step. However, many such operations are redundant. In this paper, we present a systematic approach to remove unnecessary sorting and duplicate elimination operations in path expressions in XQuery 1.0. The technique uses an automaton-based algorithm which we have applied successfully to path expressions within a complete XQuery implementation. Experimental results show that the algorithm detects and eliminates most redundant sorting and duplicate elimination operators and is very effective on common XQuery path expressions.

1 Introduction

XML is an inherently ordered data format. The relative order of elements, comments, processing instructions, and text in an XML document is significant. This *document order* makes XML an ideal format to represent information in which the order is semantically meaningful. For instance, document order can be used to represent the order in which sentences are written in a book, to represent the order of events in the report of a surgical procedure, and to represent the order in which events occurred in a log file. In addition to document order, XQuery 1.0 [3] also provides expressions to impose or access order within a sequence of items. This *sequence order* can be used for dealing with order in the traditional SQL sense. For instance, an XQuery expression can sort a sequence of persons based on their salary and order a sequence of log messages based on their size. Because of the importance of document order for many applications, the semantics of XPath requires that every intermediate step in a path expression return

* Philippe Michiels and Roel Vercammen are supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant numbers 31016 and 31581.

** Jérôme Siméon completed part of this work while at Lucent Technologies – Bell Labs.

its result in document order and without duplicates. Uniformly sorting and removing duplicates after each step, however, is expensive, therefore eliminating these operations when they are provably unnecessary can improve performance while preserving the required XPath semantics.

This work is motivated by our earlier theoretical results [11] in which we propose an efficient automaton-based approach to decide whether or not a path expression, evaluated *without* intermediate sorting and duplicate-removal operations returns a result in document order and/or without duplicates. However, this technique permits duplicates in the intermediate results, which can have a deleterious effect on performance, because all subsequent steps in the path expression are evaluated redundantly on each duplicate node. Furthermore, the presented automaton could not be used for path expression with intermediate sorting and duplicate removal operations. In this paper, we present a similar automaton-based solution to the problem of inferring document order and no duplicates in path expressions, but we do so in the context of a complete XQuery 1.0 implementation and we solve all the above problems.

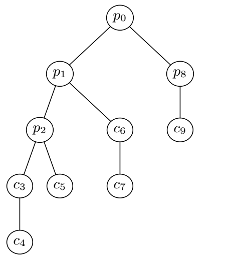
Our contribution is complementary to the research on special-purpose algorithms that support a small subset of the XPath language, in particular, steps (i.e., axis, name-test pairs) in straight-line paths [1] or trees [4]. Such algorithms use specific indices for efficient evaluation of straight-line paths and trees, and they typically do not require any intermediate sorting or duplicate elimination operators. *None* of these algorithms, however, support the complete XPath language, which permits applying arbitrary predicates to any step and querying relative document order. Moreover, each algorithm has its own limitation, such as applying to a subset of XPath axes, prohibiting wildcards, among others. Such limitations make applying these algorithms difficult in a complete XQuery implementation, which interleaves path expressions with other expressions. In contrast, our technique applies to path expressions anywhere they occur within the complete XQuery language, making it directly applicable in industrial-strength implementations.

The paper is organized as follows. In Section 2, we introduce the problem of detecting ordered and duplicate-free results in XQuery path expressions and apply our approach to an example. Section 3 describes our automaton algorithm that computes the ordered and duplicate-free properties. Insufficient space prevents presentation of all the algorithm's details. The interested reader is referred to the compete algorithm and supporting proofs in a companion technical report [6] and to the algorithm's implementation in the Galax engine [4]. Section 4 describes how our algorithm is implemented in the Galax engine. In Section 5, we report the results of applying our algorithm, called DDO, to benchmark queries. We conclude with a summary of related research in Section 6.

[4] `http://www.galaxquery.org`

2 Motivating Example

In this section, we present a simple path expression applied to the document in Fig. 1 to illustrate the problem of removing unnecessary sorting and duplicate-removal operations and our solution to this problem in XQuery. This document conforms to the partial DTD in Fig. 1, which describes Java classes and packages[5]. The tree representation of the document depicts a part of the JDK with four packages (p) and six classes (c). Integer subscripts denote document order.



```
<!ELEMENT package (package|interface|class)*>
<!ATTLIST package name CDATA #REQUIRED
    id ID #REQUIRED>
<!ELEMENT class (super, implement*,
                 (class|attribute|method)*)>
<!ATTLIST class name CDATA #REQUIRED
    id ID #REQUIRED
    visibility (public|private|protected) #IMPLIED>
```

Fig. 1. A tree representation of java classes and packages and its partial DTD

XPath queries can be applied to this document to discover interesting *code smells* (i.e., typical patterns indicating bad design), compute software metrics, specify refactorings, etc. For example, the following query returns all inner classes inside XML packages[6] (there are 67 such classes in JDK 1.4.2):

```
$jdk/desc-or-self::package[@name="xml"]/desc::class/child::class
```

In order to explain the DDO optimization on this example, we first need a notion of an *evaluation plan*. We use the XQuery Core language [5] as a simple kind of evaluation plan. The XQuery Core is easy to understand by anyone familiar with XQuery, but it also happens to be a good representation on which to apply the DDO optimization. We start with the evaluation plan obtained by applying the normalization rules in [5] to the given XQuery expression. For instance, if we simplify the previous path expression by omitting the predicate on the second step then the corresponding (simplified) XQuery core expression is shown in Fig. 2 (a).

Each step is evaluated with respect to an implicit *context node*, which is bound to the variable `$fs:dot`[7]. The `distinct-docorder` function sorts its input in document order and removes duplicates. We call such evaluation plans *tidy*, because they maintain document order and duplicate-freeness throughout evaluation of the path expression. Tidiness guarantees that the plan always yields the correct result. Moreover, intermediate results cannot grow arbitrarily large,

[5] This DTD is only a small part of the DTD we use for storing the Java Standard Library (JDK) into an XML format which is an extension of JavaML[2].

[6] Note that the attribute `name` of a `package` is not the fully qualified name of the package and hence there are several packages with name "`xml`".

[7] The `fs` namespace stands for "Formal Semantics".

```
distinct-docorder(
  for $fs:dot in distinct-docorder(
    for $fs:dot in distinct-docorder(
      for $fs:dot in $jdk
      return desc-or-self::package
    ) return desc::class)
  ) return child:class
)
```

(a)

```
docorder(
  for $fs:dot in distinct-docorder(
    for $fs:dot in (
      for $fs:dot in $jdk
      return desc-or-self::package
    ) return desc::class)
  ) return child:class
)
```

(b)

Fig. 2. Tidy (a) and duptidy (b) evaluation plan for the (simplified) example query.

because duplicates are always eliminated. Tidy evaluation plans, however, are rather expensive to evaluate. An alternative to tidy evaluation plans are *sloppy* evaluation plans resulting from the approach in [11], in which there are *no* intervening sorting or duplicate-removal operators between the steps of an evaluation plan and the `distinct-docorder` operator is only applied to the final result. We call such evaluation plans *sloppy.* Although the sloppy approach is simple and often effective, it does not always lead to the most efficient evaluation plan and, in particular, it sometimes causes an exponential explosion of the size of intermediate results. In this paper, we show how the sloppy approach can be extended such that it can handle evaluation plans in which there are sorting and duplicate-removal operations applied to intermediate results and can precisely determine which of these operations can or cannot be removed safely. Evaluation plans that are the result of this new approach are called *duptidy* evaluation plans.

3 The DDO Optimization

In this section, we present in detail the DDO algorithm for deciding the `ord` (in document order) and `nodup` (contains no duplicates) properties. We also describe how this information can be used to optimize the evaluation plan.

To make reasoning easier, we introduce a concise and more abstract notation for evaluation plans in which they are represented as lists of axes and sorting and duplicate-elimination operations. The symbols in Table 1 denote the axes. The σ denotes a sorting operation (`docorder`) and δ denotes the linear-time removal of duplicates in a sorted sequence (`distinct`). The function `distinct-docorder` is written in the abstract evaluation plan as the composition of `docorder` and `distinct`. For example, the original query evaluation plan of Section 2 is rep-

Axis Name	**Axis Symbol**	**Axis Name**	**Axis Symbol**
`child`	$\downarrow$	`parent`	$\uparrow$
`descendant`	$\downarrow^+$	`ancestor`	$\uparrow^+$
`descendant-or-self`	$\downarrow^*$	`ancestor-or-self`	$\uparrow^*$
`following`	$\twoheadrightarrow$	`preceding`	$\twoheadleftarrow$
`following-sibling`	$\dot{\twoheadrightarrow}$	`preceding-sibling`	$\dot{\twoheadleftarrow}$

Table 1. Axis names and symbols

resented by the abstract evaluation plan $\downarrow^*; \sigma; \delta; \downarrow^+; \sigma; \delta; \downarrow; \sigma; \delta$. The abstract evaluation plan is read from left to right, where each symbol denotes a step in the computation. If we omit certain operations from an evaluation plan, then the corresponding steps in the abstract notation are omitted. For example, the optimized (duptidy) query evaluation plan of Section 2 is denoted by $\downarrow^*; \downarrow^+; \sigma; \delta; \downarrow; \sigma$.

We make two assumptions: (1) an abstract evaluation plan is always evaluated against a single node (in our example, this is p_0 which is bound to `$jdk`) and (2) the evaluation of the axis functions against a single node always yields a sequence sorted in document order without duplicates. Under these two assumptions, the two evaluation plans above are equivalent in that they always yield the *same* final result. Moreover, the sizes of the intermediate results after each axis and any following σ and δ are also always the same, i.e., the intermediate results never contain duplicates.

The computation of a minimal duptidy evaluation plan corresponds to the following decision problem: given an abstract evaluation plan in which each axis step is followed by $\sigma; \delta$, determine the maximal sets of σs and δs that can be removed such that the resulting evaluation plan applied to any XML document (1) yields the same result as that of the original evaluation plan, and (2) for each step expression, the input is duplicate-free. The second abstract evaluation plan satisfies these constraints. This problem can be reduced to the problem of deciding for a given abstract evaluation plan whether its result is always sorted in document order and whether its result is always without duplicates. We denote these properties by `ord` and `nodup`, respectively. Formally, the property `ord` (`nodup`) holds for an abstract evaluation plan q if for any node n in any XML document, the result of applying q to n is in document order (duplicate free). We denote this fact as q : `ord` (q : `nodup`).

The algorithm for deciding `ord` and `nodup` is based upon inference rules that derive the `ord` and `nodup` properties. Ideally, we would like inference rules of the following form: if the abstract evaluation plan q has certain properties, then $q; s$ (with s an axis symbol, σ or δ) also has certain properties. Such rules provide an efficient way to derive the `ord` and `nodup` properties. For example, for σ and δ, the following rules hold:

$$\frac{}{q;\sigma : \texttt{ord}} \qquad \frac{q : \texttt{ord}}{q;\delta : \texttt{nodup}} \qquad \frac{q : \texttt{ord}}{q;\delta : \texttt{ord}}$$

The problem of inferring `ord` and `nodup` can be solved by introducing auxiliary properties and inference rules. To illustrate this, we present a small subset of the relevant properties and inference rules here. We refer the reader to the technical report [6] for the the complete set of inference rules and their proofs.

The auxiliary property `no2d` (for "no two distinct") holds for an abstract evaluation plan if its result never contains two distinct nodes. If `no2d` and `nodup` both hold, then there is always at most one node in the result, which implies that each extension of evaluation plan with one axis step is `ord` and `nodup`. Because of its importance, the conjunction of `no2d` and `nodup` is referred to as the `max1` property, i.e., q : `max1` iff q : `no2d` and q : `nodup`.

Another auxiliary property is `lin` (for "linear"), which holds for an abstract evaluation plan if it always holds that all two distinct nodes in the result have an ancestor-descendant relationship. The `lin` property is interesting, because if it holds, then the `parent` axis preserves the `nodup` property:

$$\frac{q : \texttt{lin}, \texttt{nodup}}{q; \uparrow : \texttt{nodup}}$$

Properties such as `no2d` and `lin` are called *set properties* because they only restrict the set of nodes in the result and do not refer to the sequence order or the presence of duplicates. Obviously, σ and δ preserve these properties since they do not change the result set. Another interesting characteristic of these properties is that they can be lost after one or more `child` axes, but are recovered if followed by the same number of `parent` axes. For example, if q : `lin` then it follows that $q; \downarrow; \uparrow$: `lin`, $q; \downarrow; \downarrow; \uparrow; \uparrow$: `lin`, $q; \downarrow; \downarrow; \downarrow; \uparrow; \uparrow; \uparrow$: `lin`, et cetera. The fact that a certain property π (re)appears after i `parent` steps is itself an interesting property and denoted by π_i. We also use the notation $\pi_{\leq j}$ as a shorthand for the conjunction of $\pi_0, \ldots, \pi_j$.

From the inference rules, we can construct a deterministic automaton that decides whether `ord` and/or `nodup` hold for abstract evaluation plans. The automaton serves two purposes: It is used to prove completeness of the inference rules for `ord` and `nodup`, and it provides an efficient algorithm for deciding these properties, which serves as the foundation for an implementation. For the completeness proof we refer the reader to technical report [6]. The automaton is described as an infinite state machine, but it can be shown that the automaton can be described by a one-counter machine and therefore permits an algorithm that operates in linear time and logarithmic space.

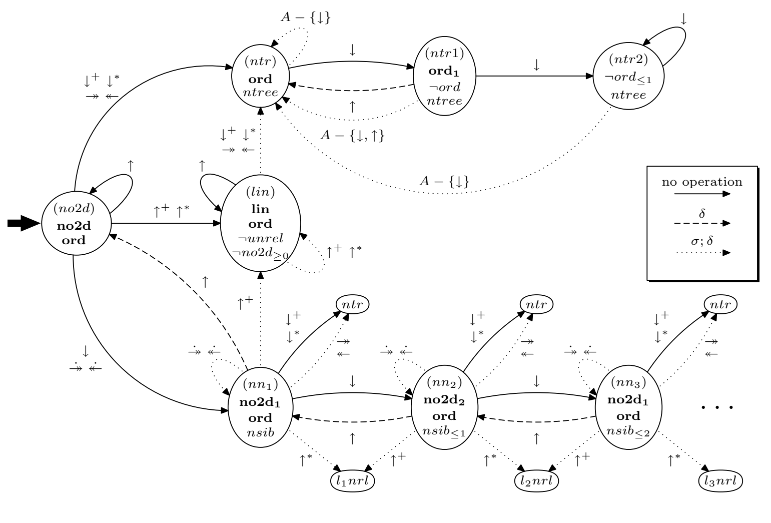


Fig. 3. Initial fragment of the duptidy automaton

Fig. 3 contains a small fragment of the complete automaton, which can be found in the technical report [6]. The initial state is the left-most state labeled *(no2d)*. The regular states contain a name (between brackets) and a list of properties that hold evaluations plans that end in this state[8] that hold . The smaller states that contain a single name with no brackets denote states for which the transitions are given in another fragment. For all regular states, a transition is given for each axis and this transition is indicated as either a solid arrow to indicate that no σ or δ is required after this step, a dashed arrow to indicate that δ is required or a dotted arrow to indicate that $\sigma;\delta$ is required. An edge labeled with $A - \{\uparrow, \downarrow\}$ indicates transitions for all axes except $\uparrow$ and $\downarrow$. It can be shown that if we construct an evaluation plan with the σs and δs as prescribed by this automaton then it is a minimal duptidy evaluation plan.

4 Implementation

We have implemented our algorithm, as well as the tidy and sloppy approaches, in the Galax XQuery engine. Galax is a complete implementation of the XQuery 1.0 specifications [3] and relies on a simple, well-documented architecture.

In our context, normalization introduces operations to sort by document order and remove duplicates after each path step, resulting in an expression that is tidy as defined in Section 3.

The DDO optimization is applied during the query-rewriting phase. As explained in Section 3, the `max1` property is required by the DDO automaton's start state. There are two ways to infer `max1`. If static typing is available and enabled then the `max1` property is derived precisely from the cardinality of the type computed for each subexpression. Static typing, however, is an optional feature of XQuery, and few implementations support it. Implementations that do not support static typing can use a simple static analysis to derive `max1`. For example, the `fn:doc` and `fn:root` functions, the `fs:dot` variable, and all variables bound by for expressions always have the `max1` property. Our algorithm starts whenever the max1 property is derived and injects `distinct` operators when intermediate results may contain duplicates and `docorder` when the final result may be out of order.

The final rewritten expression is passed to the compiler, which constructs a compiled evaluation plan for the expression and passes it to the evaluation phase. Compilation and query planning are outside the scope of this paper.

5 Experimental Results

To evaluate the impact of the proposed techniques, we conducted several experiments, the goals of which are to show that the DDO optimization is effective on common queries, and that the new *duptidy* automaton is an improvement over

[8] Note that no state contains the `nodup` property, because it holds trivially for all duptidy evaluation plans.

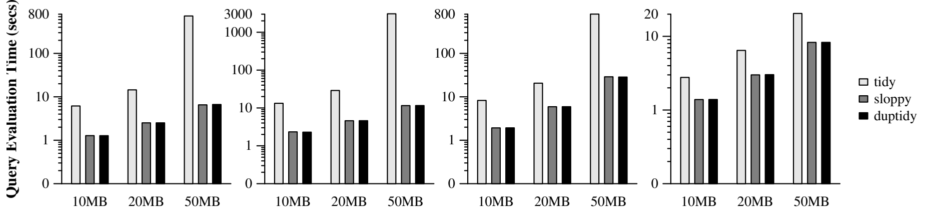


Fig. 4. The impact of the DDO-optimization for XMark queries 6, 7, 14 and 19, respectively, evaluated against input documents of size 10MB, 20MB and 50MB.

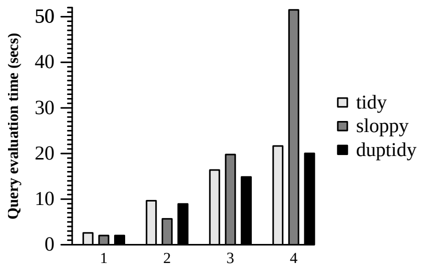
the *sloppy* automaton presented in [11]. The section is organized accordingly. All experiments have been conducted using our Galax implementation.

The first set of experiments include the familiar XMark benchmark [13] suite, applied to documents of various sizes. XMark consists of twenty queries over a document containing auctions, bidders, and items. The queries exercise most of XQuery's features (selection, aggregation, grouping, joins, and element construction, etc.) and all contain at least one path expression. Of the 239 `distinct-docorder` operations in all the normalized XMark queries, only three `docorder` operations remain after the DDO optimization.

Our optimization has the biggest impact on queries with more complex path expressions such as 6, 7, 14 and 19. Fig. 4 shows the increased impact[9] of the DDO optimization on the evaluation times of these queries as the input document increases from 10 to 50 MB. On a 20 MB document we see that Query 6 runs 5.75 times faster with the optimization and Queries 7, 14 and 19 show speedups of more than two to six times. All these queries use of the descendant-or-self axis frequently, which typically yields large intermediate results that are expensive to sort. These queries show also a bigger speedup for larger input documents. For example, the speedup for Query 7 grows from 5.79 times for a 10 MB document 10 MB to over 6 times for 20 MB to 265.33 times (!) for 50 MB. This is not surprising, because in Query 7, the evaluation time is dominated by the unnecessary sorting operations. If more nodes are selected, the relative impact of these sorting operations on the evaluation time increases.

For most of the XMark benchmark suite in Fig. 4, the sloppy approach is as effective as the duptidy approach, because none of the path expressions in these benchmarks generate duplicates in intermediate steps. For some queries, however, the sloppy technique scales poorly in the size of the path expression. This is caused by duplicate nodes in intermediate results; if duplicates are not removed immediately, subsequent steps of the path expression are applied redundantly to the same nodes multiple times. Moreover, if subsequent steps also generate duplicates, then the size of intermediate results grows exponentially. The two graphs in Fig. 5 show the results for path expressions of the form $(*//)^n*$ with $n = 1, 2, 3, 4$ applied to 52 MB and 169 MB documents, respectively. The evaluation time increases exponentially in the number of `//` steps with the sloppy

[9] Note that the Y-axes on the graphs are plotted in log scale.



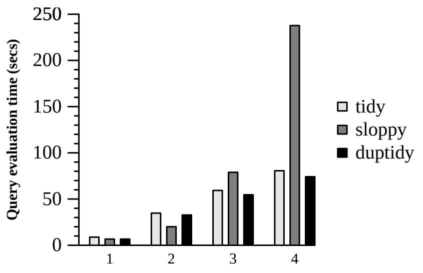


Fig. 5. Comparison of the evaluation times for the *tidy*, *sloppy* and *duptidy* approaches on the `descendant-or-self` queries for input documents of 52 MB and 169 MB.

approach. This shows that the duptidy approach combines the scalability in the size of the document of the sloppy approach with the scalability in the size of the path expression of the tidy approach. One may argue that this problem may only occur sporadically, but when it does, the impact on performance will very likely be unacceptable.

6 Related Work and Discussion

Numerous papers address the semantics and efficient evaluation of path expressions. Gottlob et al [7] show that naive implementations are often unnecessarily unscalable. Many papers deal with sorting and duplicate elimination, which is a strong indication of the importance of this problem. Avoiding duplicate elimination and sorting is particularly important in streaming evaluation strategies [12]. Helmer et al [10] present an evaluation technique that avoids the generation of duplicates, which is crucial for pipelining steps of a path expression. Grust [8, 9] proposes a similar but more holistic approach, which uses a preorder and postorder numbering for XML documents to accelerate the evaluation of path location steps in XML-enabled relational databases. By pruning the context list, the generation of duplicates and out-of-order nodes in the intermediate results is avoided, clearing the way for full pipelining. The same holds for the similar *structural join* algorithms [1] that also can compute step expressions efficiently and return a sorted result. Finally, there are also algorithms like *holistic twig joins* [4] that compute the result of multiple steps at once.

Most of this work, however, supports narrow subsets of path expressions. In contrast, our techniques apply to path expressions in the complete XQuery language. Note that the DDO optimization does not impede the above optimizations and, in fact, opens the way for a broader search space for optimizing the query evaluation plan.

Aside from that, the completeness of our approach ensures optimal results for a considerable part of the language. More precisely, it removes a maximal amount of sorting and duplicate-elimination operations from normalized path expressions under the restriction that we only allow duptidy evaluation plans. Summarizing, the DDO optimization is a relatively simple technique that finds a solution that is optimal in a certain theoretical sense and that is hard to improve upon without using more involved cost-based techniques.

In future work, we will continue to improve logical rewritings early in the compilation pipeline as well as implement more sophisticated evaluation strategies later in the compilation pipeline. Currently, the DDO optimization is limited to path expressions within XQuery, but we plan to extend the technique to all of XQuery. Simple improvements include propagating properties computed within a path expression to subsequent uses of the expression, e.g., `let`-bound variables, across function calls, etc., and using static typing properties (e.g., `max1`) not just for the head of the path expression, but for intermediate steps. As part of ongoing research we plan to implement efficient axis-evaluation strategies, such as pipeline-enabling algorithms, which are enabled by the DDO optimization.

References

1. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Pate, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE 2002*.
2. G. J. Badros. JavaML: a markup language for Java source code. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*, pages 159–177. North-Holland Publishing Co., 2000.
3. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, W3C working draft 12 november 2003, Nov 2003. `http://www.w3.org/TR/2003/WD-xquery-20031112/`.
4. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD 2002*.
5. 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, W3C working draft, Feb 2004. `http://www.w3.org/TR/2004/WD-xquery-semantics-20040220/`.
6. M. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. Automata for Avoiding Unnecessary Ordering Operations in XPath Implementations. Technical Report UA 2004-02, 2004. `http://www.adrem.ua.ac.be/pub/TR2004-02.pdf`.
7. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB 2002*.
8. T. Grust. Accelerating XPath location steps. In *SIGMOD 2002*.
9. T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *VLDB 2003*.
10. S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In *WISE 2002*.
11. J. Hidders and P. Michiels. Avoiding unnecessary ordering operations in XPath. In *DBPL 2003*.
12. F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD/PODS 2003*.
13. A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB 2002*. `http://monetdb.cwi.nl/xml/`.