Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions

Jan Hidders
University of Antwerp
jan.hidders@ua.ac.be

Philippe Michiels
University of Antwerp
philippe.michiels@ua.ac.be

Roel Vercammen*
University of Antwerp
roel.vercammen@ua.ac.be

Abstract

The semantics of the standard XML query language XQuery requires that the results of its path expressions are in document order and duplicate-free. Many implementations of this semantics guarantee correctness by inserting explicit operations that sort and remove duplicates in their evaluation plans. The sorting and duplicate-removal operations are often either inserted after each step or only after the last step. However, both strategies have performance drawbacks. In this paper we show how to create more efficient evaluation plans by deciding statically where such operations are required. We present inference rules for deciding orderedness and duplicate-freeness of the results of evaluation plans and show that these rules are sound and, for certain evaluation plans, complete. These inference rules are implemented by an efficient, automaton-based algorithm. Experimental results show that the algorithm is effective on many common path expressions.

1 Introduction

Unlike the classical relational database systems, XML databases are ordered, i.e., all its data collections are lists rather than sets. This creates a whole series of challenges and opportunities in the area of query optimization. The fact that order is so important to XML is reflected in its query languages, which usually enforce some kind of orderedness in their results even though it essentially represents a set. An example of this are path expressions in XQuery which are used to select sets of nodes in document trees but whose semantics is defined such that the result is a list of nodes that contains no duplicate nodes and orders the nodes as they appear in the document. A naive interpretation of these semantics leads to query evaluation plans that contain sorting operations (and also duplicate-elimination operations) for every step in the path expression and thereby often become performance bottlenecks and impede certain optimizations such as pipe-lined evaluation techniques. However, simply removing them for all intermediate steps can also cause performance problems since this will allow duplicates in intermediate results and thereby duplicate computations. In this work we propose a solution that uses the fact that the path expressions navigate in an ordered tree and will therefore sometimes, after navigating in certain combinations of directions, produce a correct result without an explicit sorting or duplicate-elimination operation. We present an algorithm that statically decides which of these operations are redundant and thereby allows us to determine a query evaluation plan that contains a minimal number of such operations.

1.1 Introduction to XQuery

XQuery [3] is the standard query language for XML and it is currently being standardized by the World Wide Web Consortium. The two main syntactical constructs of the language are FLWOR expressions and path expressions.

Path expressions are used to *select* nodes from the input document tree with UNIX-like path expressions. There are ten axes that allow navigation in a certain direction through the XML tree. For example, the path expression

```
doc("input.xml")/descendant::a/child::b
```

opens an XML document and selects all elements labeled *b* that are children of the *a*-labeled descendants of the document root. This path expression contains two axes, namely `descendant` and `child`. A single navigation operation is called a *step* and a path expression consists of an arbitrary number of steps. The label tests (e.g. `::a`) are called node tests. Additional conditions can be applied to the nodes that are selected by a step expression by placing conditional expressions in square brackets after the step. For instance,

```
doc("input.xml")/child::a[child::b]
```

selects those *a*-labeled children of the document root that do have *b*-children. Path expressions return ordered sequences of items. If these items are nodes selected from a document, then the nodes in the result sequence must occur in the same order as they do in the document, i.e. they must be in document order.

FLWOR , pronounced “flower”, is short hand for For - Let - Where - Order By - Return. It allows iteration over item sequences (e.g. results of path expressions) with `for`-loops. `let`-bindings are used for additional selections. The sequences can be reordered on some value with `order by` clauses and finally the result is returned in the `return` statement. A simple example of a FLWOR expression is

```
for $x in doc("input.xml")/employee
let $year := $x/employeeSince
where $x/salary > 5000
order by $year
return
  <entry name="{$x/name}" since="{$year}"/>
```

This expression selects employees and for each of them selects the year they started working for the company. The employees with a salary greater than `5000` are ordered by that year and for each of them, an element with their name and starting year is constructed and returned.

1.2 XQuery Semantics of Path Expressions

The problem that is being tackled in this paper originates from the semantics of path expressions, which are specified in terms of a limited subset of the XQuery language [5], called the XQuery Core. Consider the afore-mentioned simple path expression.

```
doc("input.xml")/descendant::a/child::b
```

The exact semantics of this expression is specified in terms of XQuery FLWOR expressions as follows (simplified):

```
distinct-docorder(
  let $sequence :=
    distinct-docorder(
      let $sequence := doc("input.xml")
      return
        for $dot in $sequence
        return descendant::a
    )
  return
    for $dot in $sequence
    return child::b
)
```

As mentioned earlier, the XQuery semantics requires the result of every path expression to be in document order. This is why the `distinct-docorder` operations (`ddo` in short) are inserted after each step. To see why this is necessary, let us apply the above expression to the XML document[1] in Figure 1.

```
<?xml version="1.0"?>
<a>
  <b/>
  <a>
    <b/>
  </a>
  <b/>
</a>
```

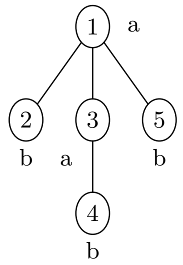


Figure 1: An XML document and its corresponding tree representation. The numbers inside the tree nodes correspond to the document order.

We see that after the first `descendant::a`-step, we get the nodes `1` and `3`, in that order, since the axis operations always return their results in document order. Evaluating the next step consists of iterating over these nodes, and selecting all the `b`-children of every node. This results in the elements `2` and `5` (for `1`) and `4` (for `3`), again in that order. Obviously, the sequence `2, 5, 4` is not in order and thus, a sorting operation has to be inserted.

So for each of the step expressions, a sorting and duplicate-elimination operation is applied, even if this may not be necessary. The most important reason for this is that path expressions can turn out to be very complex and the easiest way to get the semantics right is to sort and remove duplicates after every step. The explicit insertion of `ddo` operations causes two problems:

1. When large input documents are queried, the `ddo`-calls will have to sort very large sets of nodes.

2. The `ddo` operations are pipeline breakers. Having to materialize a large amount of nodes in memory will often slow down query evaluation and increase memory usage [7].

Therefore we would like to get rid of as many `ddo` operations as possible without giving in on correctness. A trivial and naive approach to solving this problem is to postpone sorting and duplicate elimination until the end. This is what we will refer to as the *sloppy* approach. However, this comes at the risk of an exponential blow-up of duplicates in the intermediate result, which causes a multitude of duplicate

[1] Note that <a/> is an abbreviation for <a>

computations in subsequent steps [9], which can have an unacceptable impact on evaluation performance.

Another possible approach is to remove sorting/duplicate-elimination operations from the query plan if the result is already sorted/without duplicates. In the following sections we show that this is possible by reasoning over certain static properties of path expressions with the help of relatively simple inference rules.

2 Abstract Evaluation Plans

For making reasoning easier, we make some simplifying assumptions and formalize some concepts. Aside from a formal notion for XML documents and document order, we also introduce abstract evaluation plans. These are an abstraction of the XQuery Core expressions that were mentioned earlier. Abstract evaluation plans allow us to concisely express path expressions that solely consist of navigational axis steps that do not have node tests.

We begin with the formalization of an XML document. To save space, we only consider element nodes (N). Other types of nodes can be added easily. Since node tests cannot appear in evaluation plans, we do not model node labels in our formalization of an XML document.

Definition 2.1 (XML Document)**.** *An* XML document *is a rooted ordered tree* $D = (N, \lhd, r, \prec)$ *such that* $(N, \lhd, r)$ *is a rooted tree with nodes* N*, edges* $\lhd \subseteq N \times N$ *that indicate the parent-child relation and root* r*, and* $\prec$ *is the sibling-order relation that is a strict partial order over* N *such that for each two distinct nodes* $n_1, n_2 \in N$ *it holds that* $n_1 \prec n_2$ *or* $n_2 \prec n_1$ *iff they are siblings.*

The relations $\lhd^+$ and $\lhd^*$ denote the transitive closure and the reflexive and transitive closure of $\lhd$, respectively. The reverses of $\prec$, $\lhd$, $\lhd^+$ and $\lhd^*$ are denoted by $\succ$, $\rhd$, $\rhd^+$ and $\rhd^*$, respectively. The composition of two binary relations R and S is $R \circ S = \{(n_1, n_3) | (n_1, n_2) \in S, (n_2, n_3) \in R\}$.

Next, we need to formalize the document order of nodes in an XML document.

Definition 2.2 (Document Order)**.** *Given an XML document* $D = (N, \lhd, r, \prec)$ *we define the* document order in D*,* $\ll_D$*, as the strict total order over* N *that orders the nodes as encountered in a pre-order tree-walk, i.e.,* $\ll_D = \lhd^+ \cup (\rhd^* \circ \prec \circ \lhd^*)$.

Before we define abstract evaluation plans, we first define a *set semantics* and a *sequence semantics* for each axis in terms of the above relations on nodes.

Definition 2.3 (Axes)**.** *The* set of axes A *is defined as* $\{\uparrow, \downarrow, \uparrow^+, \downarrow^+, \uparrow^*, \downarrow^*, \twoheadleftarrow, \twoheadrightarrow, \dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\}$ *where these symbols represent the axes as given in Table 1. The concise notation in Table 1 extends the notation in [2].*

Axis Name	Axis Symbol	Set Semantics $\{\!\!\{\text{Axis}\}\!\!\}_D$
`child`	$\downarrow$	$\lhd$
`parent`	$\uparrow$	$\rhd$
`descendant`	$\downarrow^+$	$\lhd^+$
`ancestor`	$\uparrow^+$	$\rhd^+$
`descendant-or-self`	$\downarrow^*$	$\lhd^*$
`ancestor-or-self`	$\uparrow^*$	$\rhd^*$
`following`	$\twoheadrightarrow$	$\rhd^* \circ \prec \circ \lhd^*$
`preceding`	$\twoheadleftarrow$	$\rhd^* \circ \succ \circ \lhd^*$
`following-sibling`	$\dot{\twoheadrightarrow}$	$\prec$
`preceding-sibling`	$\dot{\twoheadleftarrow}$	$\succ$

Table 1: Axis names, symbols, and set semantics

Definition 2.4 (Axis Set Semantics)**.** *The* set semantics *of an axis a on a document $D = (N, \lhd, r, \prec)$ is a binary relation $\{\!\!\{a\}\!\!\}_D \subseteq N \times N$ and is defined by the third column of Table 1.*

For example, the semantics of the `following` axis is defined such that it contains the pair (n_1, n_2) iff n_2 is the descendant (or the node itself) of a node that is a following sibling of an ancestor (or the node itself) of n_1.

Definition 2.5 (Axis Sequence Semantics)**.** *The* sequence semantics *of an axis a on a document $D = (N, \lhd, r, \prec)$ is a function $[\![a]\!]_D : N \to \mathcal{S}(N)$ where $\mathcal{S}(N)$ denotes the set of finite sequences over N such that $[\![a]\!]_D(n)$ is the sequence that is obtained by sorting the set $\{n'|(n, n') \in \{\!\!\{a\}\!\!\}_D\}$ with $\ll_D$, the document order of D.*

We overload the last notation and define a function $[\![a]\!]_D : \mathcal{S}(N) \to \mathcal{S}(N)$ such that it holds that $[\![a]\!]_D(\langle n_1, \ldots, n_k\rangle) = [\![a]\!]_D(n_1)\cdot\ldots\cdot[\![a]\!]_D(n_k)$ where $\cdot$ denotes sequence concatenation.

To make reasoning about evaluation plans easier, we introduce a more concise abstract notation for them. We represent evaluation plans as lists of axis symbols from Table 1 and the symbols σ and δ that represent sorting and duplicate-elimination operations, respectively. To be precise, σ represents the operation that sorts the input sequence in document order and δ represents the duplicate-elimination operation that assumes that its input is sorted in document order, which implies that it can do this in linear time and constant space. The concrete function `distinct-docorder` is always written in the abstract evaluation plan as the composition of `docorder` and `distinct`.

Definition 2.6 (Abstract Evaluation Plan)**.** *An* abstract evaluation plan *is a non-empty sequence $q = s_1; \ldots; s_k$ where each s_i is either an axis symbol, σ or δ.*

Now that we know what an abstract evaluation plan is, we need to define its semantics. We also specify when two evaluation plans are equivalent and what a correct evaluation plan is.

Definition 2.7 (Evaluation Plan Semantics)**.** *Given a document D, the* semantics of an abstract evaluation plan $q = s_1; \ldots; s_k$*, is a partial*[2] *function* $[\![q]\!]_D : N \to \mathcal{S}(N)$ *such that* $[\![q]\!]_D(n) = F(\langle n \rangle)$ *where* $F = [\![s_1]\!]_D \circ \ldots \circ [\![s_k]\!]_D$*.*

Two abstract evaluation plans are called *equivalent* if they have the same semantics. In accordance with the formal semantics of XQuery an evaluation plan $q = s_1; \ldots; s_n$ is called *correct* if it is equivalent with $s_1; \ldots; s_n; \sigma; \delta$.

Duptidy Evaluation Plans The purpose of our work is to avoid unnecessary sorting and/or duplicate eliminations to occur during the evaluation of a path expression. This boils down to finding a correct abstract evaluation plan with a minimal number of σ's and δ's. As mentioned earlier, postponing sorting and duplicate elimination until after the last step is not a viable solution, since duplicates in the intermediate result can cause an exponential blowup, both in memory usage and in execution time. Intermediate orderedness however is something we are not interested in. Thus, a nice approach would be to postpone sorting and duplicate elimination except when duplicates are introduced. If this happens, the intermediate result is sorted (if necessary) and freed from duplicates. If after the last step, the result is not sorted by document order, a final sorting operation is applied.

Such evaluation plans in which the generation of duplicates is avoided are called duptidy evaluation plans, i.e., they tidily correspond to the formal semantics with respect to duplicates in the intermediate results.

Definition 2.8 (Duptidy Evaluation Plan)**.** *A duptidy evaluation plan is a correct evaluation plan* $q = s_1; \ldots; s_n$ *such that for each XML document D and* s_i *of q that is an axis step, it holds that the range of the function* $[\![s_1; \ldots; s_{i-1}]\!]_D$ *does not contain a sequence with duplicate nodes.*

Our goal will be to decide which steps in an abstract evaluation plan produce a result which is sorted by document order/free from duplicates, independent of the the input given to the corresponding path expression. In this way, we will be able to construct "minimal" duptidy evaluation plans for path expressions, i.e., a duptidy evaluation plan that contains all the axes of the path expression in the same order and that has a minimal number of σs and δs.

3 Evaluation Plan Properties

For each step in the abstract evaluation plan we need to infer two static properties. The first property is *ord* or the orderedness property. If we can infer this property for a step, then we know its result will always be in document order, no matter which input is used to evaluate the path expression. The second property is the *nodup* property, indicating that the result of a step will always be free from duplicates. The inference of these properties allows us to remove the corresponding operations from the evaluation plan.

[2]The semantics of an evaluation plan is a partial function, because δ assumes an ordered input sequence.

Definition 3.1 (The *ord* and *nodup* Properties)**.** *For an evaluation plan q we define the following properties:*

ord (Ordered) *For every XML document D and node n_1 in D the sequence $[\![q]\!]_D(n_1)$ is sorted in the document order of D.*

nodup (No Duplicates) *For every XML document D and node n_1 in D the sequence $[\![q]\!]_D(n_1)$ contains no duplicates.*

The fact that a certain property π holds for an evaluation plan q is denoted as $q : \pi$. For example, $\downarrow;\downarrow$: *ord* denotes the fact that the result of the evaluation plan $\downarrow;\downarrow$ is always sorted in document order.

Unfortunately, the *ord* and *nodup* properties are insufficient for a complete inference mechanism and we will need additional properties. For example, consider the abstract evaluation plan $\downarrow;\downarrow$. We always assume the input cardinality of the first step to be one. We know that if we follow the child axis twice from one node, that the result will always be sorted by document order and free from duplicates. However, following the child axis does not necessarily mean that order is maintained. If the child axis is preceded by the descendant axis, for instance, then the final result can be out of document order (see the example of Section 1.1). So the fact that the second child step in $\downarrow;\downarrow$ preserves document order is due to another property that holds after the first step, namely the *unrel* property, which states that there are no ancestor-descendant related nodes in the result (see below).

Many of the additional properties are *set properties* in the sense that they only refer to the result set of the evaluation plan and do not care about the order or the multiplicity of the nodes in the result. It is clear that this is not true for the *ord* and *nodup* properties. The set properties are divided in *positive set properties* and *negative set properties*.

3.1 Positive Set Properties

Positive set properties are those set properties that forbid certain combinations of nodes in the result of the evaluation plan. For instance, the earlier mentioned *unrel* property does not allow for any two nodes in the result to be ancestor-descendant related.

Definition 3.2 (Positive Set Properties)**.** *For evaluation plans q we define the following properties:*

lin (Linear) *For every XML document D and node n_1 in D all the nodes in $[\![q]\!]_D(n_1)$ are ancestor-descendent related.*

unrel (Unrelated) *For every XML document D and node n_1 in D all the nodes in $[\![q]\!]_D(n_1)$ are* not *ancestor-descendant related.*

nolc (No Left Child) *For every XML document D, node n_1 in D and nodes n_2, n_3 in $[\![q]\!]_D(n_1)$ it holds that if n_2 has a sibling n_4 that is an ancestor of n_3 then n_2 is not a left sibling of n_4.*

norc (No Right Child) *For every XML document D, node* n_1 *in D and nodes* n_2, n_3 *in* $[\![q]\!]_D(n_1)$ *it holds that if* n_2 *has a sibling* n_4 *that is an ancestor of* n_3 *then* n_2 *is not a right sibling of* n_4.

no2d (No 2 Distinct Nodes) *For every XML document D and node* n_1 *in D there are not two distinct nodes in* $[\![q]\!]_D(n_1)$.

3.2 Negative Set Properties

Negative properties are those set properties that require that certain combinations can occur in the result of the evaluation plan.

Definition 3.3 (Negative Set Properties)**.** *For evaluation plans q we define the following properties:*

nsib (*n* Siblings) *For any number n there is an XML document D and a node* n_1 *in D such that there are at least n distinct siblings in* $[\![q]\!]_D(n_1)$.

ntree (Tree of size *n*) *For any number n there is an XML document D and a node* n_1 *in D such that there is set of nodes in* $[\![q]\!]_D(n_1)$ *that spans a tree in D of height n with all internal nodes having n children.*

nhat (Hat of *n* Siblings) *For any number n there is an XML document D and a node* n_1 *in D such that there is a node* n_2 *in* $[\![q]\!]_D(n_1)$ *such that there is an ancestor of* n_2 *which has at least n distinct left siblings and n distinct right siblings in* $[\![q]\!]_D(n_1)$.

3.3 Indexed properties

For many properties π it holds that if for an evaluation plan q it holds that $q : \pi$ then the same property also holds for q extended with $\downarrow; \uparrow$, i.e., $q; \downarrow; \uparrow : \pi$. For example this holds for the *ord* property, but not for the *nodup* property. More generally, if we extend q with i times the $\downarrow$ axis followed by i times the $\uparrow$ axis, many properties that hold for q also hold for the extended query. Therefore, we introduce indexed versions of all the properties that indicate that the original property is obtained if we apply the $\uparrow$ axis i times.

Definition 3.4 (Indexed Properties)**.** *All properties* π *except nodup can have indices such as in* π_i, $\pi_{\leq i}$ *and* $\pi_{\geq i}$, *which are defined as follows:*

- $q : \pi_0$ *iff* $q : \pi$
- *if* $i > 0$ *then* $q : \pi_i$ *iff* $(q; \uparrow) : \pi_{i-1}$.
- $q : \pi_{\leq i}$ *iff for all* $j \leq i$ *it holds that* $q : \pi_j$.
- $q : \pi_{\geq i}$ *iff for all* $j \geq i$ *it holds that* $q : \pi_j$.

The set Π *is the set of all evaluation plan properties that can have an index, i.e.,* $\Pi = \{ord, lin, unrel, nolc, norc, no2d, nsib, ntree, nhat, \neg ord, \neg lin, \neg unrel, \neg nolc, \neg norc, \neg no2d, \neg nsib, \neg ntree, \neg nhat\}$.

For all properties π we use π, π_0 and $\pi_{\leq 0}$ as synonyms. It is easy to see that if π is a positive (negative) set property then so are π_i, $\pi_{\leq i}$ and $\pi_{\geq i}$. When the property is negated *and* has an index with $\geq$ or $\leq$ then the negation is assumed to have the higher priority. So, for example, $q : \neg\pi_{\geq i}$ means that for all $j \geq i$ it holds that $q : \neg\pi_j$.

3.4 Negated properties

The fact that a certain property does *not* hold for a certain evaluation plan is also relevant since, for example, we want to be able to derive when *ord* holds and when it does not. Therefore we introduce negated versions of all properties π which are written as $\neg\pi$. The semantics of $q : \neg\pi$ is then assumed to be that it does not hold that $q : \pi$. So, if $q : \neg ord$ then it is not true that the result of q is always sorted. Note that this does not mean the the result is always unsorted. Also note that it is easy to see that the negated version of a positive set property becomes a negative set property and vice versa.

4 Inference Rules

We are now ready to define our inference mechanism. We will define a set of inference rules for deriving properties for abstract evaluation plans. Our goal is to be complete for abstract evaluation plans in the sense that we want to be able to derive for each step of every evaluation plan that either *ord* or $\neg$*ord* holds, and likewise for *nodup*. In this paper however we only consider rules that are needed to be complete for duptidy evaluation plans. Table 2 shows the complete set of rules for duptidy evaluation plans.

Below we discuss some of these rules in more detail. We will also show some example proofs for a few rules. Due to spacing constraints we refer the reader to the technical report [6] for the other proofs and the entire set inference rules that is complete for all abstract evaluation plans.

Rule 8 The first rule we discuss tell us that if an evaluation plan q has the *unrel* and *nodup* properties, following any downward axis will preserve the *nodup* property.

$$\frac{q : unrel, nodup \qquad a \in \{\downarrow, \downarrow^*, \downarrow^+\}}{q; a : nodup}$$

Proof. By definition of a tree, two unrelated nodes never have common descendants. Therefore, unrelated nodes will never generate duplicates when one of the axes $\downarrow$, $\downarrow^+$ or $\downarrow^*$ are followed. □

Rule 2 The next rule states that if the result of an evaluation plan q is always in document order and q has the *norc* property, then the result of $q;\uparrow$ is also always in document order.

$$\frac{q : norc, ord}{q;\uparrow\, : ord}$$

Proof. Let a and b be two output nodes of q with $a \ll$[3] b and let c and d be their respective parent nodes in the result of the step expression. Suppose now that $d \ll c$. This means that either d is an ancestor of c or d is a preceding node of c.

- d **is a preceding node of** c – If this were true, then all children of d, including b, would be preceding nodes of c. This however conflicts with the fact that $a \ll b$;

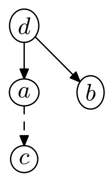
- d **is an ancestor node of** c – Since $a \ll b$, this implies that there is an ancestor of c that has a right sibling, namely b. This however conflicts with the definition of the *norc* property.

We conclude that any two parents of two input nodes occur in order in the result. □

Rule 35 To see the importance of the *norc* property, we now consider the same rule, with the difference that the *norc* property does *not* hold. We show that if this is the case, the ord property is not preserved.

$$\frac{q : \neg norc, ord}{q;\uparrow\, : \neg ord}$$

Proof. The $\neg$*norc* property implies that there is a document such that after following q the input sequence contains two nodes b and c, which are structured as follows:



Solid arrows indicate a child-parent relationship and dashed arrows indicate an ancestor-descendant relationship. Since $c \ll b$ and the *ord* property holds for the input, we know that c comes before b in the input sequence. Suppose the parent of c is e. Then e comes before d (which is the parent of b) in the output sequence, but from the fact that e is a descendant of d follows that $d \ll e$. Hence the output sequence is not in document order. □

[3] From now on we write $\ll$ instead of $\ll_D$ when it is clear for which document D we want to express the document order.

Rule 76 The last rule states that if there are at least two different nodes of the same tree in the input, then following any of the recursive axes $\downarrow^*, \downarrow^+, \uparrow^*, \uparrow^+, \twoheadleftarrow, \twoheadrightarrow$, causes the output to have the $\neg ord_{\geq 1}$ property, which means that following the parent axis one or more times from there, results in a node sequence that can be out of document order.

$$\frac{q : \neg no2d \qquad a \in \{\downarrow^*, \downarrow^+, \uparrow^*, \uparrow^+, \twoheadleftarrow, \twoheadrightarrow\}}{q; a : \neg ord_{\geq 1}}$$

Proof. **[For $\downarrow^+$, $\downarrow^*$, $\twoheadrightarrow$, $\twoheadleftarrow$]** After following any one of these axes, rule 41 says that the *ntree* property holds. The rules 65 and 57 allow us to deduce that $q : \neg norc_{\geq 0}$. So, $q; \uparrow^n$ has both the *ord* and the $\neg norc$ property and thus $q; \uparrow^n$ has the $\neg ord_1$ property. The result now contains two related nodes that are out of document order and it is easy to see that this remains the case after following any number of parent axes.

[For $\uparrow^+$, $\uparrow^*$] The sequences that result from following these axes from two different nodes share an arbitrarily long subsequence of nodes. The result sequence thus contains two identical subsequences of arbitrary length, which implies that there can be two related nodes that are not in document order, no matter how many times you follow the parent axis. □

$$(1)\ \frac{q : no2d, nodup \qquad a \in A}{q; a : ord} \qquad (2)\ \frac{q : norc, ord}{q; \uparrow : ord} \qquad (3)\ \frac{\begin{array}{l} q : unrel, ord, nodup \\ a \in \{\downarrow, \downarrow^*, \downarrow^+\} \end{array}}{q; a : ord}$$

$$(4)\ \frac{q : lin, ord}{q; \dot{\twoheadleftarrow} : ord} \qquad (5)\ \frac{q : no2d, nodup \qquad a \in A}{q; a : nodup} \qquad (6)\ \frac{q : nodup}{q; \downarrow : nodup}$$

$$(7)\ \frac{\begin{array}{l} q : lin, nodup \\ a \in \{\uparrow, \dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\} \end{array}}{q; a : nodup} \qquad (8)\ \frac{\begin{array}{l} q : unrel, nodup \\ a \in \{\downarrow, \downarrow^*, \downarrow^+\} \end{array}}{q; a : nodup} \qquad (9)\ \frac{q : no2d_n \qquad n \geq 0}{q : no2d_{n+1}}$$

$$(10)\ \frac{q : no2d_n \qquad n \geq 0}{q : lin_n} \qquad (11)\ \frac{q : lin_n \qquad n \geq 0}{q : lin_{n+1}} \qquad (12)\ \frac{q : lin_n \qquad n \geq 1}{q; \uparrow^+ : lin_{n-1}}$$

$$(13)\ \frac{q : lin_n \qquad n \geq 0}{q; \uparrow^* : lin_n} \qquad (14)\ \frac{q : lin}{q : nolc} \qquad (15)\ \frac{q : nolc}{q; \dot{\twoheadrightarrow} : nolc}$$

$$(16)\ \frac{q : nolc_n, lin_{n+1} \qquad n \geq 0}{q; \uparrow^* : nolc_n} \qquad (17)\ \frac{q : nolc_n, lin_{n+1} \qquad n \geq 1}{q; \uparrow^+ : nolc_{n-1}} \qquad (18)\ \frac{q : no2d_n \qquad n \geq 0}{q : nolc_{\geq 0}}$$

$$(19)\ \frac{q : lin}{q : norc} \qquad (20)\ \frac{q : norc}{q; \dot{\twoheadleftarrow} : norc} \qquad (21)\ \frac{\begin{array}{l} q : norc_n, lin_{n+1} \\ n \geq 0 \end{array}}{q; \uparrow^* : norc_n}$$

$$(22)\ \frac{q : norc_n, lin_{n+1} \qquad n \geq 1}{q; \uparrow^+ : norc_{n-1}} \qquad (23)\ \frac{q : unrel_1 \qquad a \in \{\dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\}}{q; a : norc} \qquad (24)\ \frac{q : unrel}{q; \downarrow : norc}$$

$$(25)\quad \frac{q : \mathit{no2d}_n \qquad n \geq 0}{q : \mathit{norc}_{\geq 0}}$$

$$(26)\quad \frac{q : \mathit{nolc}}{q; \dot{\twoheadrightarrow} : \mathit{unrel}}$$

$$(27)\quad \frac{q : \mathit{norc}}{q; \dot{\twoheadleftarrow} : \mathit{unrel}}$$

$$(28)\quad \frac{q : \mathit{unrel}_n \qquad n \geq 1}{q : \mathit{unrel}_{n-1}}$$

$$(29)\quad \frac{q : \mathit{no2d}_n \qquad n \geq 0}{q : \mathit{unrel}}$$

$$(30)\quad \frac{\begin{array}{l} \pi \in \Pi \qquad q : \pi_n \\ n \geq 0 \end{array}}{q; \downarrow : \pi_{n+1}}$$

$$(31)\quad \frac{\begin{array}{l} \pi \in \Pi \qquad q : \pi_n \\ n \geq 1 \end{array}}{q; \uparrow : \pi_{n-1}}$$

$$(32)\quad \frac{\begin{array}{l} \pi \in \Pi \qquad q : \pi_n \\ a \in \{\dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\} \qquad n \geq 1 \end{array}}{q; a : \pi_n}$$

$$(33)\quad \frac{\begin{array}{l} q : \neg \mathit{no2d} \\ a \in \{\twoheadleftarrow, \twoheadrightarrow, \uparrow^*, \uparrow^+\} \end{array}}{q; a : \neg \mathit{ord}}$$

$$(34)\quad \frac{\begin{array}{l} q : \neg \mathit{unrel} \\ a \in \{\downarrow, \downarrow^*, \downarrow^+\} \end{array}}{q; a : \neg \mathit{ord}}$$

$$(35)\quad \frac{q : \neg \mathit{norc}, \mathit{ord}}{q; \uparrow : \neg \mathit{ord}}$$

$$(36)\quad \frac{\begin{array}{l} q : \mathit{nsib} \\ a \in \{\dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\} \end{array}}{q; a : \neg \mathit{ord}}$$

$$(37)\quad \frac{q : \neg \mathit{unrel}, \mathit{ord}}{q; \dot{\twoheadrightarrow} : \neg \mathit{ord}}$$

$$(38)\quad \frac{\begin{array}{l} q : \neg \mathit{no2d} \\ a \in \{\twoheadleftarrow, \twoheadrightarrow, \uparrow^*, \uparrow^+\} \end{array}}{q; a : \neg \mathit{nodup}}$$

$$(39)\quad \frac{\begin{array}{l} q : \neg \mathit{unrel} \\ a \in \{\downarrow^*, \downarrow^+\} \end{array}}{q; a : \neg \mathit{nodup}}$$

$$(40)\quad \frac{q : \mathit{nsib} \qquad a \in \{\uparrow, \dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\}}{q; a : \neg \mathit{nodup}}$$

$$(41)\quad \frac{a \in \{\downarrow^*, \downarrow^+, \twoheadleftarrow, \twoheadrightarrow\}}{q; a : \mathit{ntree}}$$

$$(42)\quad \frac{q : \mathit{ntree} \qquad a \in A}{q; a : \mathit{ntree}}$$

$$(43)\quad \frac{a \in \{\uparrow^*, \uparrow^+\}}{q; a : \neg \mathit{unrel}}$$

$$(44)\quad \frac{q : \neg \mathit{unrel} \qquad a \in \{\downarrow, \uparrow\}}{q; a : \neg \mathit{unrel}}$$

$$(45)\quad \frac{q : \neg \mathit{norc}}{q; \dot{\twoheadleftarrow} : \neg \mathit{unrel}}$$

$$(46)\quad \frac{q : \neg \mathit{nolc}}{q; \dot{\twoheadrightarrow} : \neg \mathit{unrel}}$$

$$(47)\quad \frac{q : \mathit{ntree}}{q : \neg \mathit{unrel}}$$

$$(48)\quad \frac{q : \mathit{lin}, \neg \mathit{no2d}}{q : \neg \mathit{unrel}}$$

$$(49)\quad \frac{a \in \{\uparrow^*, \uparrow^+\}}{q; a : \neg \mathit{no2d}_{\geq 0}}$$

$$(50)\quad \frac{q : \neg \mathit{no2d}_{\geq 0} \qquad a \in A}{q; a : \neg \mathit{no2d}_{\geq 0}}$$

$$(51)\quad \frac{q : \mathit{nsib}}{q : \neg \mathit{no2d}}$$

$$(52)\quad \frac{a \in \{\downarrow, \dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\}}{q; a : \mathit{nsib}}$$

$$(53)\quad \frac{q : \mathit{nhat}_n \qquad n \geq 0}{q : \mathit{nsib}_n}$$

$$(54)\quad \frac{q : \mathit{nhat}_n \qquad n \geq 0}{q : \neg \mathit{nolc}_n}$$

$$(55)\quad \frac{q : \neg \mathit{unrel}}{q; \dot{\twoheadleftarrow} : \neg \mathit{nolc}}$$

$$(56)\quad \frac{q : \neg \mathit{nolc}}{q; \dot{\twoheadleftarrow} : \neg \mathit{nolc}}$$

$$(57)\quad \frac{q : \mathit{nhat}_n \qquad n \geq 0}{q : \neg \mathit{norc}_n}$$

$$(58)\quad \frac{q : \neg \mathit{unrel}}{q; \dot{\twoheadrightarrow} : \neg \mathit{norc}}$$

$$(59)\quad \frac{q : \neg \mathit{norc}}{q; \dot{\twoheadrightarrow} : \neg \mathit{norc}}$$

$$(60)\quad \frac{q : \neg \mathit{unrel}}{q; \downarrow : \mathit{nhat}}$$

$$(61)\quad \frac{q : \neg \mathit{nolc}}{q; \dot{\twoheadrightarrow} : \mathit{nhat}}$$

$$(62)\quad \frac{q : \neg \mathit{norc}}{q; \dot{\twoheadleftarrow} : \mathit{nhat}}$$

$$(63)\quad \frac{q : \mathit{nsib}_n \qquad n \geq 1}{q; \uparrow^* : \mathit{nhat}_{n-1}}$$

$$(64)\quad \frac{q : \mathit{nsib}_n \qquad n \geq 2}{q; \uparrow^+ : \mathit{nhat}_{n-2}}$$

$$(65)\quad \frac{q : \mathit{ntree}}{q : \mathit{nhat}}$$

$$(66)\quad \frac{\begin{array}{l} \pi \in \{\mathit{nsib}, \mathit{nhat}, \\ \neg \mathit{nolc}, \neg \mathit{norc}\} \\ q : \pi_n \qquad n \geq 1 \end{array}}{q; \uparrow^+ : \pi_{n-1}}$$

$$\text{(67)}\quad \frac{\begin{array}{l}\pi \in \{nsib, nhat, \\ \neg nolc, \neg norc\} \\ q : \pi_n \qquad n \geq 0\end{array}}{q; \uparrow^* : \pi_n}$$

$$\text{(68)}\quad \frac{q : ord, lin}{q : ord_{\geq 0}}$$

$$\text{(69)}\quad \frac{\begin{array}{l}q : ord_n \qquad n \geq 1 \\ a \in \{\dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\}\end{array}}{q; a : ord_n}$$

$$\text{(70)}\quad \frac{\begin{array}{l}q : nsib \\ a \in \{\uparrow^+, \uparrow^*, \twoheadrightarrow, \twoheadleftarrow\}\end{array}}{q; a : \neg ord_{\geq 0}}$$

$$\text{(71)}\quad \frac{\begin{array}{l}q : \neg unrel \\ a \in \{\downarrow^*, \downarrow^+, \twoheadrightarrow, \twoheadleftarrow\}\end{array}}{q : \neg ord_{\geq 0}}$$

$$\text{(72)}\quad \frac{\begin{array}{l}q : \neg nodup \\ a \in \{\downarrow, \dot{\twoheadleftarrow}, \dot{\twoheadrightarrow}\}\end{array}}{q; a : \neg ord}$$

$$\text{(73)}\quad \frac{\begin{array}{l}q : \neg nodup \\ a \in \{\twoheadleftarrow, \twoheadrightarrow, \downarrow^*, \downarrow^+, \uparrow^*, \uparrow^+\}\end{array}}{q; a : \neg ord_{\geq 0}}$$

$$\text{(74)}\quad \frac{\begin{array}{l}q : \neg ord_n \qquad n \geq 1 \\ a \in \{\uparrow, \uparrow^*, \uparrow^+\}\end{array}}{q; a : \neg ord_{n-1}}$$

$$\text{(75)}\quad \frac{\begin{array}{l}q : \neg ord_{\leq n} \qquad n \geq 0 \\ a \in \{\downarrow, \downarrow^*, \downarrow^+\}\end{array}}{q; a : \neg ord_{\leq n+1}}$$

$$\text{(76)}\quad \frac{\begin{array}{l}q : \neg no2d \\ a \in \{\downarrow^*, \downarrow^+, \uparrow^*, \uparrow^+, \twoheadleftarrow, \twoheadrightarrow\}\end{array}}{q; a : \neg ord_{\geq 1}}$$

$$\text{(77)}\quad \frac{q : \neg nodup \qquad a \in A}{q; a : \neg nodup}$$

$$\text{(78)}\quad \frac{\begin{array}{l}\pi \text{ is a set property} \\ q : \pi \qquad a \in \{\sigma, \delta\}\end{array}}{q; a : \pi}$$

Table 2: All inference rules for duptidy evaluation plans

5 The Duptidy Automaton

The above inference rules allow us to construct an infinite deterministic automaton[4] that decides whether *ord* and/or *nodup* hold for abstract evaluation plans. Because the automaton is rather large, it is spread over several pages. Figures 2,3, 4 and 5 show the entire automaton.

The initial state is the lower-left state labeled *(no2d)* in Figure 2. The regular states contain a name (between brackets) and a list of properties. The smaller pseudo-states with a single name without brackets refer to real states for which transitions are given in another part of the drawing. For all regular states, a transition is given for each axis and this transition is indicated as either a solid, a dashed or a dotted arrow. An edge labeled with $A - \{\uparrow, \downarrow\}$ indicates transitions for all axes except $\uparrow$ and $\downarrow$. Although not shown in the drawing we assume that each state also contains the *nodup* property. This is because this automaton will only deal with *duptidy* evaluation plans and the properties refer to properties of intermediate results before the next axis step. For such intermediate results the *nodup* property clearly always holds.

[4]The automaton can be implemented as a finite pushdown automaton, but we prefer this representation for clearness reasons.

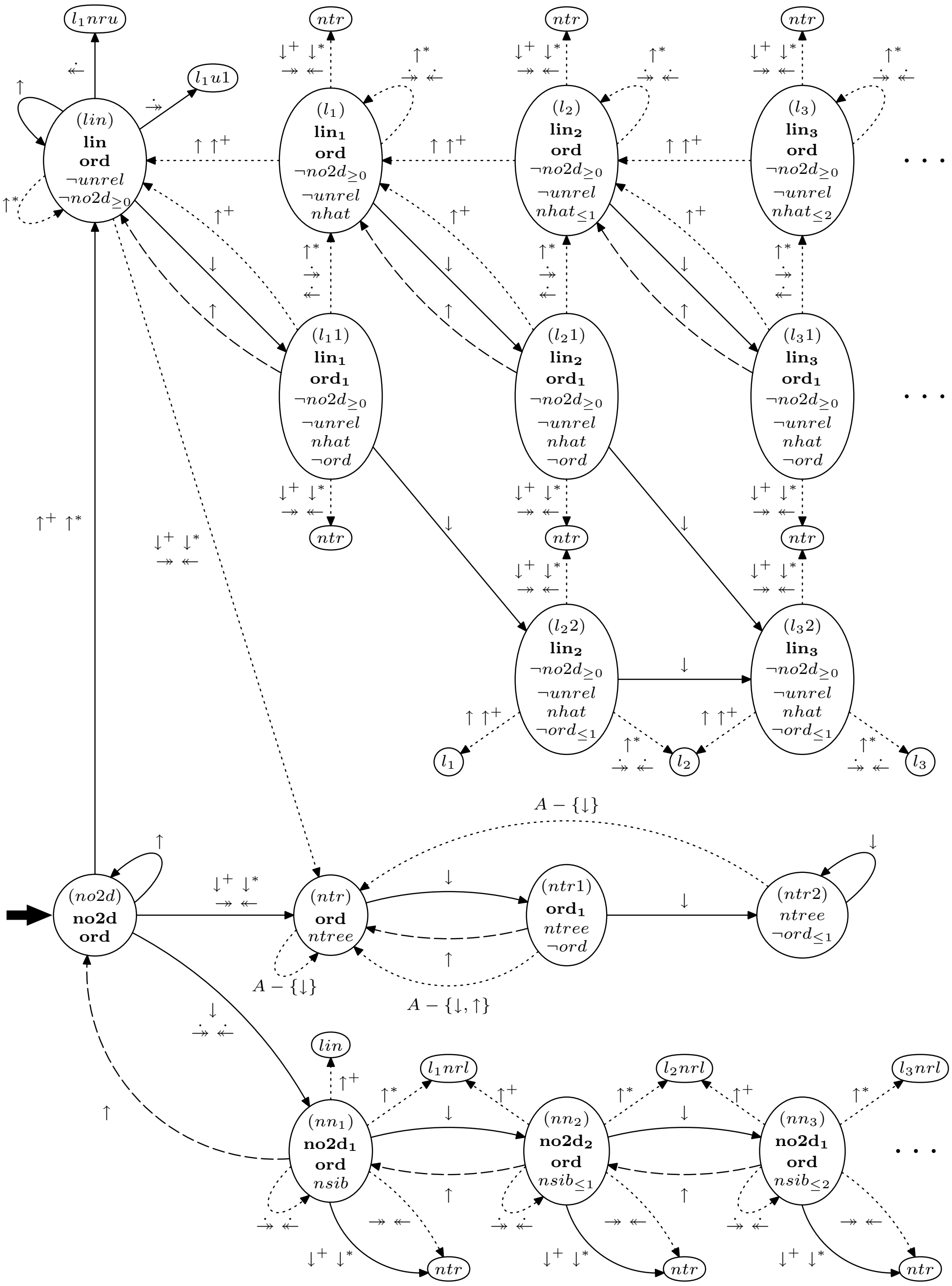


Figure 2: The Automaton $A^{duptidy}$ (Part I)

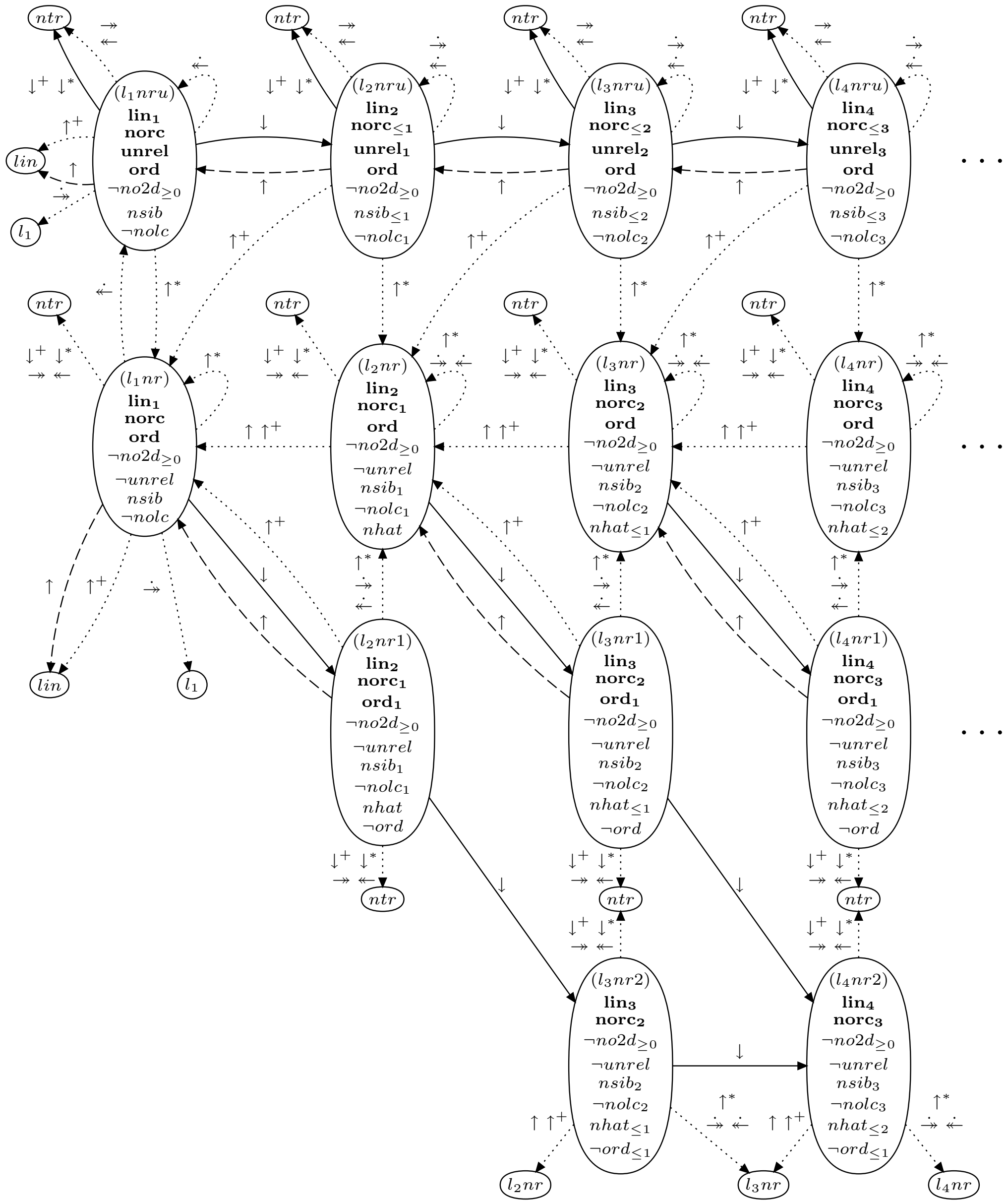


Figure 3: The Automaton $A^{duptidy}$ (Part II)

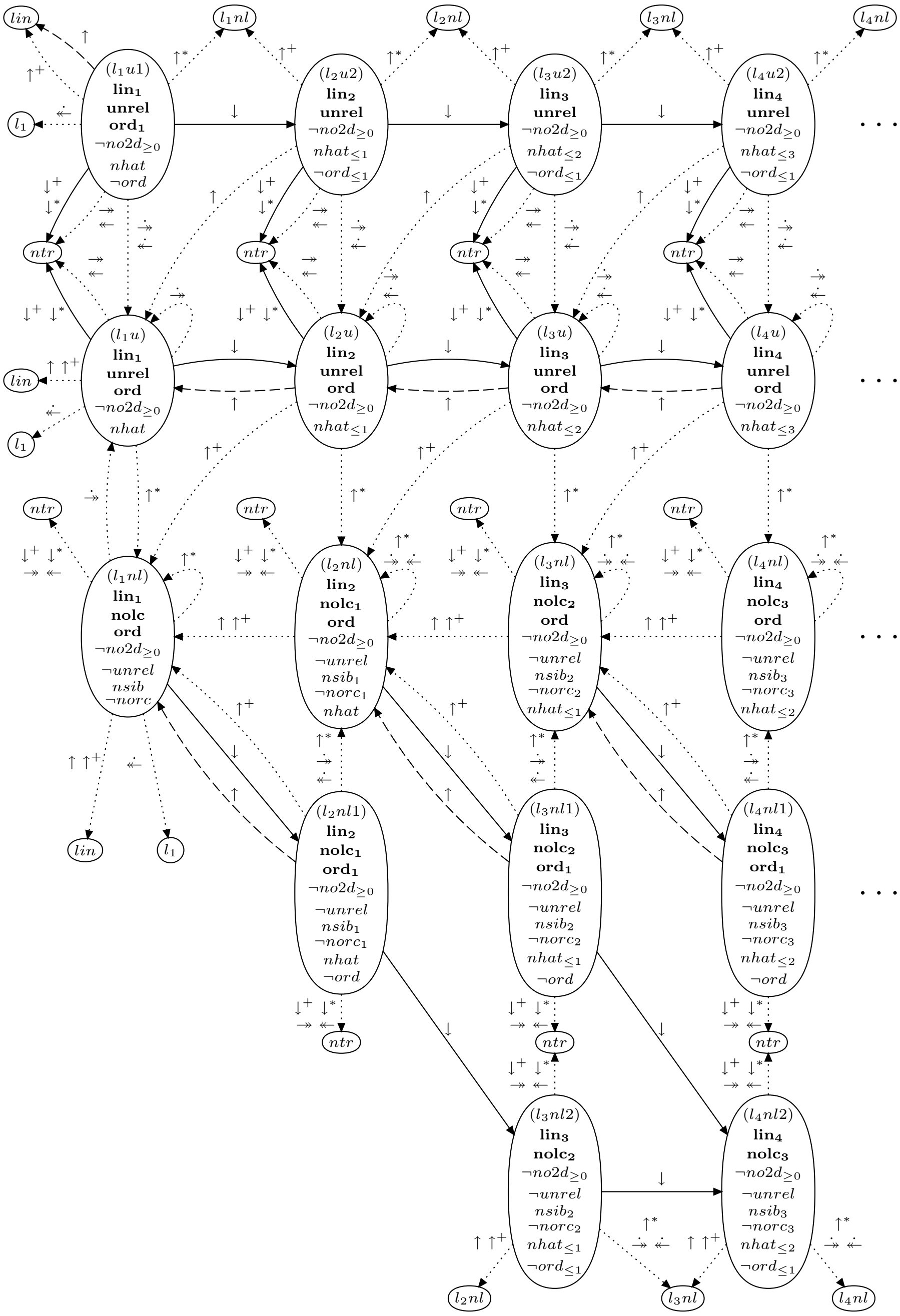


Figure 4: The Automaton $A^{duptidy}$ (Part III)

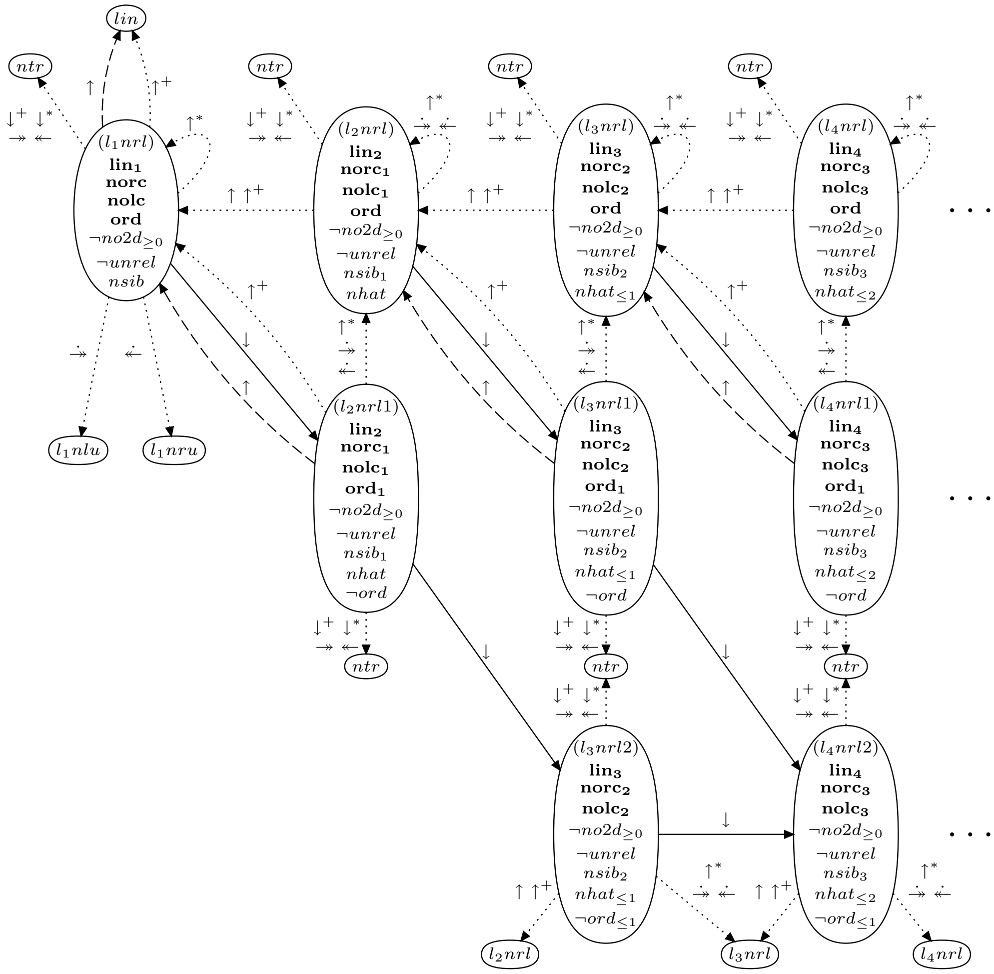


Figure 5: The Automaton $A^{duptidy}$ (Part IV)

5.1 Meaning of the Automaton

The automaton allows us to derive static properties of certain evaluation plans. However, it works only correct for evaluation plans that are automaton-correct which is defined as follows: an abstract evaluation plan is *automaton-correct* if it holds for each axis step in the plan that if the corresponding edge in the automaton is dashed or dotted then it is followed by δ or $\sigma;\delta$ respectively. For such evaluation plans it then holds that if we start from the initial state and then follow the transitions that correspond to the axis steps as they are encountered in the evaluation plan, then the properties in the state in which we end hold for this evaluation plan. Moreover, the type of the last edge indicates whether after the last axis step, *nodup* and *ord* hold or not. A solid edge indicates that *nodup* holds; a dashed edge indicates that *ord* and ¬*nodup* hold; and a dotted edge indicates that ¬*ord* and ¬*nodup* hold.

To illustrate this, consider the following abstract evaluation plan: $\downarrow^+; \dot{\twoheadrightarrow}; \sigma; \delta; \downarrow; \uparrow; \delta$. If we follow the corresponding transitions in the automaton in Figure 2, we see that this is indeed a automaton-correct evaluation plan since the $\downarrow^+$ transition from *no2d* to *ntr* is solid, the $\dot{\twoheadrightarrow}$ transition from *ntr* to *ntr* is dotted, the $\downarrow$ transition from *ntr* to *ntr1* is solid and finally the $\uparrow$ transition from *ntr1* to *ntr* is dashed. We can then conclude from the types of arrows that right after the $\downarrow^+$ step, *nodup* holds; right after the $\dot{\twoheadrightarrow}$ step, neither *ord* nor *nodup* right after the $\downarrow$ step, *nodup* holds; and finally, right after the $\uparrow$ step, *ord* and $\neg$*nodup* hold. From the labels of the states, we conclude that after the $\downarrow^+$ step, the properties *ord* and *ntree* hold; after $\dot{\twoheadrightarrow}; \sigma; \delta$ these properties again hold; after $\downarrow$, the properties ord_1, $\neg$*ord* and *ntree* hold; and finally, after $\uparrow; \delta$, the properties *ord* and *ntree* hold again.

The correctness of the automaton is established by the following theorem.

Theorem 1 (Automaton correctness)**.** *For every automaton-correct abstract evaluation plan p it holds that $p : \pi$ if the list of axis steps in p end in $A^{duptidy}$ in a state labeled with π. Moreover, if p' is equal to p except that the δ and σ steps after the last axis step are omitted then the following holds for the type of the corresponding edge in the automaton: if solid then $p' : nodup$, if dashed then $p' : \neg nodup, ord$ and if dotted then $p' : \neg nodup, \neg ord$.*

Sketch. This theorem can be proved by induction upon the length of p and p' and the inference rules in Table 2. □

As a corollary it follows that these inference rules are complete for *ord* and *nodup* in the sense that for each p' they derive either $p' : ord$ or $p' \neg ord$ and either $p' : nodup$ or $p' : \neg nodup$. Another consequence of the correctness of the automaton is that all automaton-correct evaluation plans are duptidy evaluation plans. This holds since in an automaton-correct evaluation plan it holds that after every axis step the intermediate result has either the *nodup* property or we find either $\sigma; \delta$ or δ immediately after it.

5.2 Usage of the Automaton

The automaton can be used to construct an efficient evaluation plan for a given list of axes as follows:

1. Determine the corresponding automaton-correct evaluation plan p, and
2. add at the end σ if the automaton derives that $p : \neg ord$.

We call the resulting evaluation plan the *automaton-based evaluation plan*. For illustration consider the following list of axes: $\downarrow; \downarrow; \uparrow; \uparrow^+; \downarrow$. The corresponding edges in the automaton are respectively solid, solid, dashed, dotted and solid. Therefore the corresponding automaton-correct evaluation plan is $\downarrow; \downarrow; \uparrow; \delta; \uparrow^+; \sigma; \delta; \downarrow$ and since the final state for the initial list of axes (state $l_1 1$) contains $\neg$*ord* we add at the end a final σ operation.

The following theorem establishes the correctness of and optimality of the constructed evaluation plan.

Theorem 2. *Given a list of axes the resulting automaton-based evaluation plan is (1) a correct and duptidy abstract evaluation plan and (2) is optimal in the sense that there is no equivalent duptidy evaluation plan that contains the same axes in the same order but has fewer σ and δ operations.*

Sketch. Let p be the automaton-based evaluation plan. From the correctness of the automaton and the way p is constructed it follows that right before every δ step in p the intermediate result is always sorted in document order and hence its result is always defined. Since the result of an automaton-correct evaluation plans never contains duplicates and σ is added if its result may become unsorted it follows that the final result of p is always sorted and without duplicates. The duptidiness of p holds since the initial automaton-correct evaluation plan is already duptidy.

The optimality of p for δ steps follows easily from the correctness of the automaton since they are inserted between axes steps iff they are needed to become duptidy. Moreover, σ steps are only inserted right before δ steps in order to ensure that their input is always sorted. Considering that the result of δ is always sorted (it assumes an ordered input) it holds in every correct evaluation plan p' that contains the same axes steps and δ steps in the same order as p that between two δ steps without an intermediate δ step there is at least one σ step iff p contains a σ step before the last δ step. □

6 Experiments

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
- *duptidy* evaluation plans perform better than those that follow the sloppy approach, i.e., postpone all sorting and duplicate elimination until the end of the evaluation plan.

The section is organized accordingly. All experiments have been conducted using our GALAX implementation and executed on an Intel Pentium 4 (2.4GHz) with 1GB main memory and a 7200 RPM disk, running Debian Linux 2.6.4.

6.1 XMark Benchmarks

We start by applying our optimization onto the XMark benchmark [15] suite, using input 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. Table 3 compares the query-evaluation times for the XMark queries executed on a 20MB input document without the optimization (tidy) and with the optimization applied (duptidy). The last column shows the relative speedup.

Query	Tidy	Duptidy	Speedup	Query	Tidy	Duptidy	Speedup
Q01	0.194	0.171	1.14	Q11	10.878	10.762	1.01
Q02	0.065	0.044	1.46	Q12	7.127	7.084	1.01
Q03	0.596	0.484	1.23	Q13	0.106	0.093	1.13
Q04	0.625	0.560	1.12	**Q14**	**20.616**	**5.922**	**3.48**
Q05	0.206	0.188	1.09	Q15	0.099	0.048	2.08
Q06	**14.463**	**2.516**	**5.75**	Q16	0.110	0.068	1.63
Q07	**28.953**	**4.608**	**6.28**	Q17	0.217	0.165	1.31
Q08	0.523	0.446	1.17	Q18	0.233	0.229	1.01
Q09	0.836	0.781	1.07	**Q19**	**6.442**	**3.019**	**2.13**
Q10	6.133	5.912	1.04	Q20	1.198	1.070	1.12

Table 3: Total query evaluation time (secs) of XMark queries on 20MB documents.

Of the 239 `distinct-docorder` operations in all the normalized XMark queries, only three `docorder` operations remain after the DDO optimization. For many XMark queries, however, the measured improvement is modest. The reasons for this modest improvement include:

- In all queries except Queries 6, 7, 14 and 19, every path expression contains only child steps, which permits very efficient evaluation, even without the optimization.

- Many of the queries apply selections that reduce the number of intermediate results, which in turn reduces the cost of sorting and duplicate removal.

- The axes used in the XMark benchmark suite are restricted to child and descendant-or-self. One of the immediate consequences is that duplicate nodes *never* occur in the intermediate results.

Despite the simplicity the path expressions in the XMark queries, Table 3 still shows some interesting results. Overall, the complete XMark test suite runs nearly twice as fast under the DDO optimization. Query 6 runs 5.75 times faster with the optimization and Queries 7, 14 and 19 show speedups of 6.28, 3.48 and 2.13, respectively. All these queries use of the descendant-or-self axis frequently, which typically yields large intermediate results that are expensive to sort.

Figure 6 shows the increased impact of the DDO optimization on query evaluation times as the input document grows from 10 to 50 MB. (Note that the Y-axis on the 50MB graph is plotted in log scale.) For instance, the speedup on Query 7 grows from 5.79 times for a 10MB document 10MB to over 6 times for 20MB to 265.33 times (!) for 50MB. This 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.

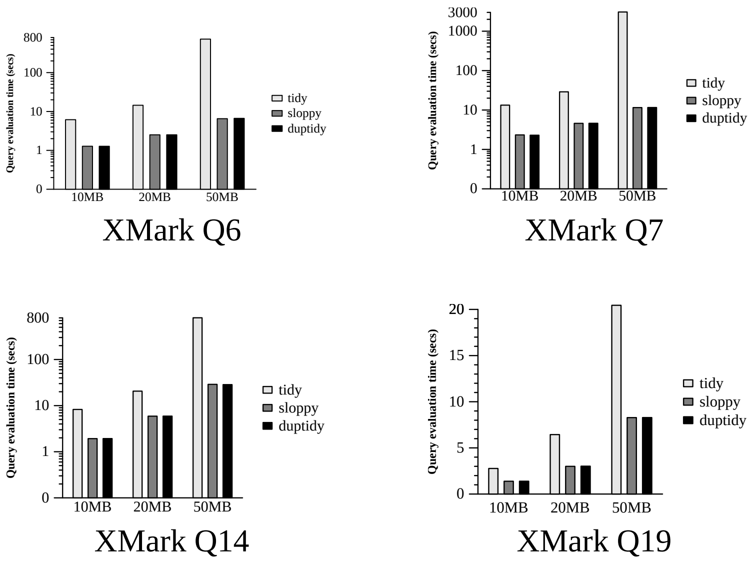


Figure 6: The impact of the DDO-optimization for XMark queries 6, 7, 14 and 19 and input documents of size 10MB, 20MB and 50MB.

6.2 Comparing Sloppy and Duptidy Evaluation Plans

As we discussed earlier in this article, the sloppy approach –although simple and easy to implement– is not always the best solution for the DDO problem. Since it allows duplicate nodes in intermediate results to propagate throughout query evaluation, they may trigger many duplicate computations for each subsequent step.

We have implemented the sloppy approach in GALAX and conducted several experiments. For many of the queries, including the entire XMark benchmark suite, this sloppy approach is as effective as the duptidy approach. Figure 6 shows the uniformity of the two approaches. For some queries, however, we observed that the sloppy technique was substantially slower, and in some cases, exponentially slower. Duplicate nodes in intermediate results are the obvious culprits. 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 intermediate results grows exponentially. To show this behavior, we evaluated path expressions of the following form for increasing values of *n* (example taken from [8]):

$$\texttt{\$input}\underbrace{\texttt{/child} :: *\texttt{/parent} :: * \ \ldots\ \texttt{/child} :: *\texttt{/parent} :: *}_{n \text{ times}}$$

We applied this expression to an input document consisting of three nodes:

```
<?xml version="1.0"?>
<node1><node2/><node3/></node1>
```

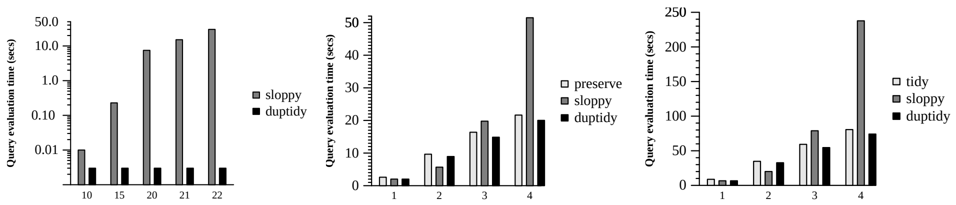


Figure 7: The graph compares the evaluation times for the *sloppy* and *duptidy* approaches on two synthetic queries for increasing lengths of step sequences. The first graph shows results for `child-parent` queries, the second and third graph show results for `descendant-or-self` queries on two different sizes of input documents.

The left-most graph in Figure 6.2 shows how the evaluation time doubles each time a child-parent step sequence is added to the above path expression. This is not surprising as the number of duplicate nodes in the intermediate result doubles every time, doubling the time to calculate subsequent steps and the time to perform the final sorting and duplicate removal operator.

The child-parent example is somewhat unusual, but a very common expression can result in similarly bad behavior. The center and right-most graphs in Figure 6.2 show the impact of multiple `//` steps in a path expression applied to 5.2 MB and 16.9 MB documents, respectively. The evaluation time increases exponentially with the sloppy approach.

The duptidy approach, in contrast, yields evaluation plans that grow gracefully with the size of the path expressions and the size of input documents, because intermediate results never contain duplicates. The duptidy approach permits intermediate results to be out of document order, but as soon as duplicates are generated, one sorting operation is left in place, if necessary, and duplicates are removed in linear. The automaton closely guards the path-expression semantics with respect to duplicates, which is why we refer to it as *duptidy*.

The left-most graph in Figure 6.2 shows how the duptidy approach causes the evaluation time to remain nearly constant, independent of how many child-parent steps are added to the path expression. In the second and third graphs, the duptidy approach scales much better with the size of both the query and the input document.

7 Related Work and Conclusion

The importance of sorting and eliminating duplicates in XQuery is underlined by the numerous papers that address this issue. The problem becomes even more prominent in streaming evaluation strategies [14]. Helmer et al [12] present an evaluation technique that avoids the generation of duplicates, which is crucial for pipelining the evaluation of path expressions. Grust [10] proposes a similar but more holistic approach for querying XML-enabled relational databases The preorder and postorder numbering of nodes is used to accelerate the evaluation of several axes by using B-tree indices. In subsequent work, Grust [11] introduces the *staircase join*, a tree-aware operator that can further speed up evaluation of path expressions. These results are ported to the XML DOM datamodel in [13] and are complementary to the approach taken in this work. 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. It is important however to point out that most of this work supports only narrow subsets of path expressions. In contrast, our techniques apply to path expressions in the complete XQuery language.

We believe that taking care of the DDO optimization is a necessary first step in any complete implementation of XQuery path expressions. The DDO optimization produces semantically correct and simplified path expressions that can be input to further query optimization. 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-eliminate operations from normalized path expressions under the restriction that we only allow duptidy evaluation plans. Note that even under this restriction the resulting evaluation plan may not be truly optimal because there can be more than one maximal set of removable operations. For example, the abstract evaluation plans $\uparrow^*; \dot{\twoheadrightarrow}; \downarrow; \sigma$ and $\uparrow^*; \dot{\twoheadrightarrow}; \sigma; \downarrow$ are both duptidy evaluation plans in which a maximal number of σs and δs were removed, but only the first one is returned by the algorithm. However, determining the most optimal one requires a cost-based approach where the estimated cost of the sorting operations depends upon the estimated sizes of the intermediate results. 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.

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 *Proceedings of the IEEE International Conference on Data Engineering*, 2002.

[2] M. Benedikt, W. Fan, and G. Kuper. Structural properties of XPath fragments. In *The 9th International Conference on Database Theory (ICDT)*, 2003.

[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 *Proceedings of the 2002 ACM SIGMOD international conference on*

Management of data, pages 310–321, Madison, Wisconsin, USA, June 2002. ACM Press.

[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 20 february 2004, 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 2004-02, University of Antwerp and AT&T Research and IBM Watson, 2004. `http://www.adrem.ua.ac.be/pub/TR2004-02.pdf`.

[7] A. GoldBerg and R. Paige. Stream processing. In *Proceedings of the ACM Symposium on LISP and Functional Programming Languages*, pages 53–62, 1984.

[8] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, Hong Kong, 2002.

[9] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Priciples of Database Systems (PODS)*, San Diego (CA), 2003.

[10] T. Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120, Madison, 2002.

[11] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *Proceedings of the 29th International Conference on Very Large Databases, VLDB'2003*, pages 524–535, Berlin, Germany, September 2003. Morgan Kaufmann Publishers.

[12] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In *Proc. of the 3rd International Conference on Web Information Systems Engineering (WISE)*, pages 215–224, Singapore, 2002.

[13] J. Hidders and P. Michiels. Efficient XPath axis evaluation for DOM data structures. In *PLAN-X*, Venice, Italy, 2004.

[14] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proc. of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Priciples of Database Systems (PODS)*, San Diego (CA), 2003.

[15] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'2002)*, pages 974–985, Hong Kong, China, 2002. `http://monetdb.cwi.nl/xml/`.