Appeared in *Proc. VLDB 2002*, Aug. 2002, Hong Kong, China. Extended version.

Efficient Algorithms for Processing XPath Queries[*]

Georg Gottlob, Christoph Koch, and Reinhard Pichler

Database and Artificial Intelligence Group
Technische Universität Wien, A-1040 Vienna, Austria
{gottlob, koch}@dbai.tuwien.ac.at, reini@logic.at

Abstract

Our experimental analysis of several popular XPath processors reveals a striking fact: Query evaluation in each of the systems requires time exponential in the size of queries in the worst case. We show that XPath can be processed much more efficiently, and propose main-memory algorithms for this problem with polynomial-time combined query evaluation complexity. Moreover, we present two fragments of XPath for which linear-time query processing algorithms exist.

1 Introduction

XPath has been proposed by the W3C [18] as a practical language for selecting nodes from XML document trees. The importance of XPath stems from (1) its potential application as an XML query language per se and it being at the core of several other XML-related technologies, such as XSLT, XPointer, and XQuery and (2) the great and well-deserved interest such technologies receive [1]. Since XPath and related technologies will be tested in ever-growing deployment scenarios, its implementations need to scale well *both* with respect to the size of the XML data and the growing size and intricacy of the queries (usually referred to as *combined complexity*).

Recently, there has been some work on related problems such as query containment for XPath [5, 10, 15], the expressiveness and complexity of various fragments of XSLT [2, 11], and contributions towards a formal semantics definition of XPath [20, 13]. However, to the best of our knowledge, no research results on good or even reasonable methods for processing XPath have been published which may serve as yardsticks for new algorithms.

[*] This work was supported by the Austrian Science Fund (FWF) under project No. Z29-INF. All methods and algorithms presented in this paper are covered by a pending patent. Further resources, updates, and possible corrections will be made available at `http://www.xmltaskforce.com`.

Contributions

In this paper, we show that it is possible to noticeably improve the efficiency of existing and future XPath engines. We claim that current implementations of XPath processors do not live up to their potential. The way XPath is defined in [18] motivates an implementation approach that leads to highly inefficient (exponential-time) XPath processing, and many implementations seem to have naively followed this intuition. Likewise, the semantics of a fragment of XPath defined in [13], which uses a fully functional formalism, motivates an exponential-time algorithm.

To get a better understanding of the state-of-the-art of XPath implementations, we experiment with three existing XPath processors, namely XALAN, XT, and Microsoft Internet Explorer 6 (IE6). XALAN [21] is a framework for processing XPath and XSLT which is freely available from the Apache foundation. XT [4] is a freely available XSLT[1] processor written by James Clark. IE6 is a commercial Web browser which supports the formatting of XML documents using XSL. Our experiments show that the time consumption of all three systems grows exponentially in the size of XPath queries in general. This exponentiality is a very practical problem. Of course, queries tend to be short, but we will argue that meaningful practical queries are *not short enough* to allow the existing systems to handle them.

The main contributions of this paper, apart from our experiments, are the following:

- We define a formal bottom-up semantics of XPath (i.e., for the full language as proposed in [18]), which leads to a bottom-up main-memory XPath processing algorithm that runs in low-degree polynomial time in terms of the data and of the query size in the worst case. By a *bottom-up algorithm* we mean a method of processing XPath while traversing the *parse tree* of the query from its leaves up to its root.
- We discuss a general mechanism for translating our

[1]Of course, XSLT allows to embed and execute arbitrary XPath queries.

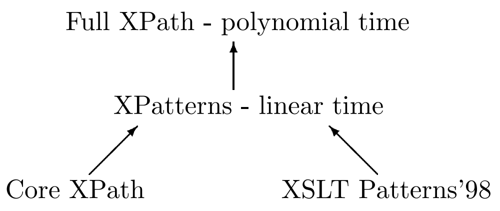


Figure 1: XPath fragments considered in this paper.

bottom-up algorithm into a top-down one. ("Top-down" again relates to the parse tree of the query.) Both have the same worst-case bound on running times but the latter may compute fewer useless intermediate results than the bottom-up algorithm.

- We present a linear-time algorithm (in both data and query size) for a practically useful fragment of XPath, which we will call *Core XPath* in the sequel.

  In the experiments presented in this paper, we show that evaluating such queries in XALAN and XT already takes exponential time in the size of the queries in the worst case. The processing time of IE6 for this fragment grows polynomially in the size of queries, but requires quadratic time in the size of the XML *data* (when the query is fixed).

- We discuss the now superseded language of *XSLT Patterns* of the XSLT draft of December 16th, 1998 [17]. Since then, full XPath has been adopted as the XSLT Pattern language. This language remains interesting, as it shares many features with XPath and is a useful practical query language. We extend this language with all of the XPath axes and call it *XPatterns* to keep it short. Surprisingly, XPatterns queries can be evaluated very efficiently, in linear time in the size of the data and the query.

The rationale for presenting these fragments is their relevance to the efficiency of engines for full XPath on common queries. An overview of the various query language fragments considered in this paper and data complexity bounds of the associated algorithms is given in Figure 1. By $\mathcal{L}_1 \leftarrow \mathcal{L}_2$, we denote that language $\mathcal{L}_1$ subsumes language $\mathcal{L}_2$: XPatterns fully subsumes the Core XPath language, and subsumes XSLT Patterns'98 (except for a minor detail). XPatterns is a fragment of XPath.

Structure

The structure of this paper is as follows. In Section 2, we provide experimental results for existing XPath processors. Section 3 introduces axes for navigation in trees. Section 4 presents the data model of XPath and auxiliary functions used throughout the paper. Section 5 defines the semantics of XPath in a concise way. Section 6 houses the bottom-up semantics definition and algorithm for full XPath, and Section 7 comes up with the modifications to obtain a top-down algorithm. Section 8 presents linear-time fragments of XPath (Core XPath and XPatterns). We conclude with Section 9.

2 State-of-the-Art of XPath Systems

In this section, we evaluate the efficiency of three XPath engines, namely Apache XALAN (the Lotus/IBM XPath implementation which has been donated to the Apache foundation) and James Clark's XT, which are, as we believe, the two most popular freely available XPath engines, and Microsoft Internet Explorer 6 (IE6), a commercial product. The reader is assumed familiar with XPath and standard notions such as *axes* and *location steps* (cf. [18]).

The version of XALAN used for the experiments was Xalan-j_2_2_D11 (i.e., a Java release). We used the current version of XT (another Java implementation) with release tag 19991105, as available on James Clark's home page, in combination with his XP parser through the SAX driver. We ran both XALAN and XT on a 360 MHz (dual processor) Ultra Sparc 60 with 512 MB of RAM running Solaris. IE6 was evaluated on a Windows 2000 machine with a 1.2 GHz AMD K7 processor and 1.5 GB of RAM.

XT and IE6 are not literally XPath engines, but are able to process XPath embedded in XSLT transformations. We used the xsl:foreach performative to obtain the set of all nodes an XPath query would evaluate to.

We show by experiments that all three implementations require time exponential in the size of the queries in the worst case. Furthermore, we show that even the simplest queries, with which IE6 can deal efficiently in the size of the queries, take quadratic time in the size of the data. Since we used two different platforms for running the benchmarks, our goal of course was not to compare the systems against each other, but to test the scalabilities of their XPath processing algorithms. The reason we used two different platforms was that Solaris allows for accurate timing, while IE6 is only available on Windows. (The IE6 timings reported on here have the precision of ± 1 second).

For our experiments, we generated simple, flat XML documents. Each document $DOC(i)$ was of the form

$$\langle \mathrm{a}\rangle \underbrace{\langle \mathrm{b}/\rangle \dots \langle \mathrm{b}/\rangle}_{i \text{ times}} \langle /\mathrm{a}\rangle$$

and its tree thus contained $i + 1$ element nodes.

Experiment 1: Exponential-time Query Complexity of XALAN and XT

In this experiment, we used the fixed document $DOC(2)$ (i.e., $\langle \mathrm{a}\rangle\langle \mathrm{b}/\rangle\langle \mathrm{b}/\rangle\langle /\mathrm{a}\rangle$). Queries were constructed using a simple pattern. The first query was '//a/b' and the $i+1$-th query was obtained by taking the i-th query and appending '/parent::a/b'. For instance, the third query was '//a/b/parent::a/b/parent::a/b'.

It is easy to see that the time measurements reported in Figure 2, which uses a log scale Y axis, grow exponentially with the size of the query. The sharp bend in the curves is due to the near-constant runtime overhead of the Java VM and of parsing the XML document.

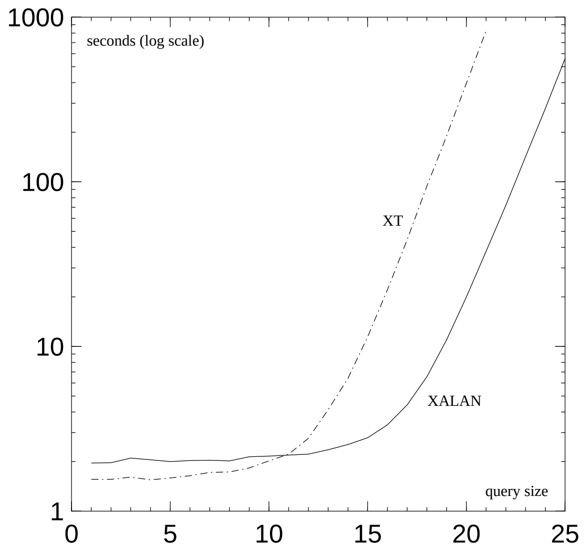


Figure 2: Exponential-time query complexity of XT and XALAN (Experiment 1).

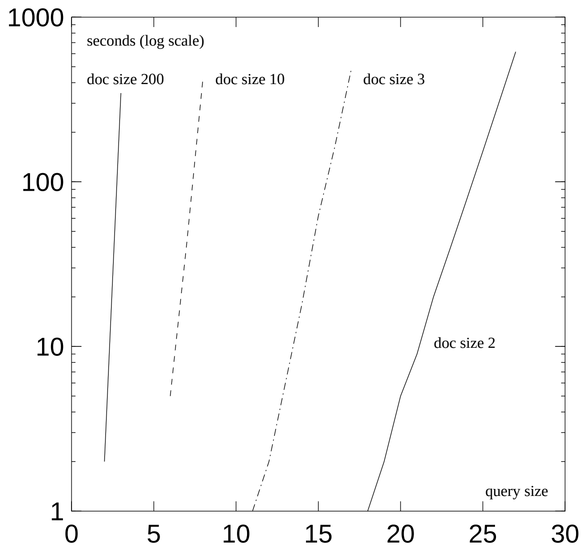


Figure 3: Exponential-time query complexity of IE6, for document sizes 2, 3, 10, and 200 (Experiment 2).

Discussion

This behavior can be explained with the following pseudocode fragment, which seems to appropriately describe the basic query evaluation strategy of XALAN and XT.

```
procedure process-location-step($n_0$, $Q$)
/* $n_0$ is the context node;
   query $Q$ is a list of location steps */
begin
   node set $S$ := apply $Q$.first to node $n_0$;
   if ($Q$.tail is not empty) then
      for each node $n \in S$ do
         process-location-step($n$, $Q$.tail);
end
```

It is clear that each application of a location step to a context node may result in a set of nodes of size linear in the size of the document (e.g., each node may have a linear number of descendants or nodes appearing after it in the document). If we now proceed by recursively applying the location steps of an XPath query to individual nodes as shown in the pseudocode procedure above, we end up consuming time exponential in the size of the query in the worst case, even for very simple path queries. As a (simplified) recurrence, we have

$$\text{Time}(|Q|) := \begin{cases} |D| * \text{Time}(|Q|-1) & \dots \quad |Q| > 0 \\ 1 & \dots \quad |Q| = 0 \end{cases}$$

where $|Q|$ is the length of the query and $|D|$ is the size of the document, or equivalently

$$\text{Time}(|Q|) = |D|^{|Q|}.$$

The class of queries used puts an emphasis on simplicity and reproducibility (using the *very* simple document ⟨a⟩⟨b/⟩⟨b/⟩⟨/a⟩). Interestingly, each 'parent::a/b' sequence quite exactly doubles the times both systems take to evaluate a query, as we first jump (back) to the tree root labeled "a" and then experience the "branching factor" of two due the two child nodes labeled "b".

Our class of queries may seem contrived; however, it is clear that we make a practical point. First, more realistic document sizes allow for very short queries only[2]. At the same time, XPath query engines need to be able to deal with increasingly sophisticated queries, along the current trend to delegate larger and larger parts of data management problems to query engines, where they can profit from their efficiency and can be made subject to optimization. The intuition that XPath can be used to match a large class of *tree patterns* [12, 9, 3] in XML documents also implies to a certain degree that queries may be extensive.

Moreover, similar queries using antagonist axes such as "following" and "preceding" instead of "child" and "parent" do have practical applications, such as when we want to put restrictions on the relative positions of nodes in a document. Finally, if we make the realistic assumption that the documents are always much larger than the queries ($|Q| << |D|$), it is not even necessary to jump back and forth with antagonist axes. We can use queries such as //following::*/following::*/.../following::* to observe exponential behavior.

Experiment 2: Exponential-time Query Complexity of Internet Explorer 6

In our second experiment, we executed queries that nest two important features of XPath, namely paths

[2]We will show this in the second experiment for IE6 (see Figure 3), and have verified it for XALAN and XT as well.

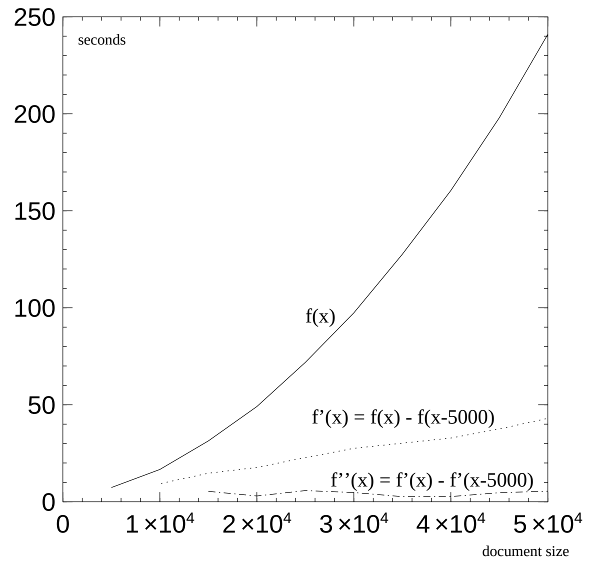


Figure 4: Quadratic-time data complexity of IE6. f' and f'' are the first and second derivatives, respectively, of our graph of timings f (Experiment 3).

and arithmetics, using IE6. The first three queries were

//a/b[count(parent::a/b) > 1]
//a/b[count(parent::a/b[
    count(parent::a/b) > 1]) > 1]
//a/b[count(parent::a/b[
    count(parent::a/b[
        count(parent::a/b) > 1]) > 1]) > 1]

and it is clear how to continue this sequence.

The experiment was carried out for four document sizes (2, 3, 10, and 200). Figure 3 shows clearly that IE6 requires time exponential in the size of the query.

Experiment 3: Quadratic-time Data Complexity for Simple Path Queries (IE6)

For our third experiment, we took a fixed query and benchmarked the time taken by IE6 for various document sizes. The query was '//a' + $q(20)$ + '//b' with

$$q(i) := \begin{cases} \text{`//b[ancestor::a'} + q(i-1) & \\ \quad + \text{`//b]/ancestor::a'} & \ldots \quad i > 0 \\ \text{`'} & \ldots \quad i = 0 \end{cases}$$

(Note: The size of queries $q(i)$ is of course $O(i)$.)

Example 2.1 For instance, the query of size two according to this scheme, i.e. '//a' + $q(2)$ + '//b', is

//a//b[ancestor::a//b[ancestor::a//b
                    ]/ancestor::a//b
    ]/ancestor::a//b □

The granularity of measurements (in terms of document size) was 5000 nodes. Figure 4 shows that IE6 takes quadratic time w.r.t. the size of the data already for this simple class of path queries.

The query complexity of IE6 w.r.t. such queries is polynomial as well. Due to space limitations, we do not provide a graph for this experiment.

By virtue of our experiments, the following question naturally arises: Is there an algorithm for processing XPath with guaranteed polynomial-time behavior (combined complexity), or even one that requires only linear time for simple queries? In the remainder of this paper, we are able to provide a positive answer to this.

3 XPath Axes

In XPath, an XML document is viewed as an unranked (i.e., nodes may have a variable number of children), ordered, and labeled tree. Before we make the data model used by XPath precise (which distinguishes between several types of tree nodes) in Section 4, we introduce the main mode of navigation in document trees employed by XPath – axes – in the abstract, ignoring node types. We will point out how to deal with different node types in Section 4.

All of the artifacts of this and the next section are defined in the context of a given XML document. Given a document tree, let dom be the set of its nodes, and let us use the two functions

$$\text{firstchild}, \text{nextsibling} : \text{dom} \rightarrow \text{dom},$$

to represent its structure[3]. "firstchild" returns the first child of a node (if there are any children, i.e., the node is not a leaf), and otherwise "null". Let $n_1, \ldots, n_k$ be the children of some node in document order. Then, $\text{nextsibling}(n_i) = n_{i+1}$, i.e., "nextsibling" returns the neighboring node to the right, if it exists, and "null" otherwise (if $i = k$). We define the functions firstchild^{-1} and nextsibling^{-1} as the inverses of the former two functions, where "null" is returned if no inverse exists for a given node. Where appropriate, we will use binary relations of the same name instead of the functions. ($\{\langle x, f(x)\rangle \mid x \in \text{dom},\ f(x) \neq \text{null}\}$ is the binary relation for function f.)

The axes *self*, *child*, *parent*, *descendant*, *ancestor*, *descendant-or-self*, *ancestor-or-self*, *following*, *preceding*, *following-sibling*, and *preceding-sibling* are binary relations $\chi \subseteq \text{dom} \times \text{dom}$. Let $\text{self} := \{\langle x, x\rangle \mid x \in \text{dom}\}$. The other axes are defined in terms of our "primitive" relations "firstchild" and "nextsibling" as shown in Table 1 (cf. [18]). $R_1.R_2$, $R_1 \cup R_2$, and R_1^* denote the concatenation, union, and reflexive and transitive closure, respectively, of binary relations R_1 and R_2. Let $E(\chi)$ denote the regular expression defining χ in Table 1. It is important to observe that some axes are defined in terms of other axes, but that these definitions are acyclic.

Definition 3.1 (Axis Function) Let χ denote an XPath axis relation. We define the function χ :

[3]Actually, "firstchild" and "nextsibling" are part of the XML Document Object Model (DOM).

child := firstchild.nextsibling*
parent := (nextsibling^{-1})*.firstchild^{-1}
descendant := firstchild.(firstchild $\cup$ nextsibling)*
ancestor := (firstchild^{-1} $\cup$ nextsibling^{-1})*.firstchild^{-1}
descendant-or-self := descendant $\cup$ self
ancestor-or-self := ancestor $\cup$ self
following := ancestor-or-self.nextsibling. nextsibling*.descendant-or-self
preceding := ancestor-or-self.nextsibling^{-1}. (nextsibling^{-1})*.descendant-or-self
following-sibling := nextsibling.nextsibling*
preceding-sibling := (nextsibling^{-1})*.nextsibling^{-1}

Table 1: Axis definitions in terms of "primitive" tree relations "firstchild", "nextsibling", and their inverses.

$2^{\text{dom}} \to 2^{\text{dom}}$ as $\chi(X_0) = \{x \mid \exists x_0 \in X_0 : x_0 \chi x\}$ (and thus overload the relation name χ), where $X_0 \subseteq$ dom is a set of nodes. □

Algorithm 3.2 (Axis Evaluation)
Input: A set of nodes S and an axis χ
Output: $\chi(S)$
Method: $\text{eval}_\chi(S)$

```
function $\text{eval}_{(R_1 \cup \ldots \cup R_n)^*}(S)$ begin
    $S' := S$;    /* $S'$ is represented as a list */
    while there is a next element $x$ in $S'$ do
        append $\{R_i(x) \mid 1 \leq i \leq n,\ R_i(x) \neq \text{null},$
                          $R_i(x) \notin S'\}$ to $S'$;
    return $S'$;
end;
function $\text{eval}_\chi(S) := \text{eval}_{E(\chi)}(S)$.
function $\text{eval}_{\text{self}}(S) := S$.
function $\text{eval}_{e_1.e_2}(S) := \text{eval}_{e_2}(\text{eval}_{e_1}(S))$.
function $\text{eval}_R(S) := \{R(x) \mid x \in S\}$.
function $\text{eval}_{\chi_1 \cup \chi_2}(S) := \text{eval}_{\chi_1}(S) \cup \text{eval}_{\chi_2}(S)$.
```

where $S \subseteq$ dom is a set of nodes of an XML document, e_1 and e_2 are regular expressions, R, R_1, ..., R_n are primitive relations, χ_1 and χ_2 are axes, and χ is an axis other than "self". □

Clearly, some axes could have been defined in a simpler way in Table 1 (e.g., ancestor equals parent.parent*). However, the definitions, which use a limited form of regular expressions only, allow to compute $\chi(S)$ in a very simple way, as evidenced by Algorithm 3.2.

Consider the directed graph $G = (V, E)$ with $V =$ dom and $E = R_1 \cup \ldots \cup R_n$. The function $\text{eval}_{(R_1 \cup \ldots \cup R_n)^*}$ essentially computes graph reachability on G (not transitive closure). It can be implemented to run in linear time in terms of the size of the data (corresponding to the edge relation E of the graph[4]) in a straightforward manner; (non)membership in S' is checked in constant time using a direct-access version of S' maintained in parallel to its list representation (naively, this could be an array of bits, one for each member of dom, telling which nodes are in S').

[4]Note that $|E| \approx 2 \cdot |T|$, where $|T|$ is the size of the edge relation of the document tree.

Lemma 3.3 *Let $S \subseteq$ dom be a set of nodes of an XML document and χ be an axis. Then,*

1. *$\chi(S) = eval_\chi(S)$ and*

2. *Algorithm 3.2 runs in time $O(|dom|)$.*

Proof Sketch ($O(|\text{dom}|)$ running time). The time bound is due to the fact that each of the eval functions can be implemented so as to visit each node at most once and the number of calls to eval functions and relations joined by union is constant (see Table 1). □

4 Data Model

Let dom be the set of nodes in the document tree as introduced in the previous section. Each node is of one of seven *types*, namely root, element, text, comment, attribute, namespace, and processing instruction. As in DOM [16], the root node of the document is the only one of type "root", and is the *parent* of the document element node of the XML document. The main type of non-terminal node is "element", the other node types are self-explaining (cf. [18]). Nodes of all types besides "text" and "comment" have a name associated with it.

A *node test* is an expression of the form $\tau()$ (where τ is a node type or the wildcard "node", matching any type) or $\tau(n)$ (where n is a node name and τ is a type whose nodes have a name). $\tau(*)$ is equivalent to $\tau()$. We define a function T which maps each node test to the subset of dom that satisfies it. For instance, $T(\text{node}()) =$ dom and $T(\text{attribute(href)})$ returns all attribute nodes labeled "href".

Example 4.1 Consider $DOC(4)$ of Section 2. It consists of *six* nodes – the document element node a labeled "a", its four children $b_1, \ldots, b_4$ (labeled "b"), and a root node r which is the parent of a. We have $T(\text{root}()) = \{r\}$, $T(\text{element}()) = \{a, b_1, \ldots, b_4\}$, $T(\text{element}(a)) = \{a\}$, and $T(\text{element}(b)) = \{b_1, \ldots, b_4\}$. □

Now, XPath axes differ from the abstract, untyped axes of Section 3 in that there are special child axes "attribute" and "namespace" which filter out all resulting nodes that are not of type attribute or namespace, respectively. In turn, all other XPath axis functions remove nodes of these two types from their results. We can express this formally as

$$\begin{aligned} \text{attribute}(S) &:= \text{child}(S) \cap T(\text{attribute}()) \\ \text{namespace}(S) &:= \text{child}(S) \cap T(\text{namespace}()) \end{aligned}$$

and for all other XPath axes χ (let χ_0 be the abstract axis of the same name),

$$\chi(S) := \chi_0(S) - (T(\text{attribute}()) \cup T(\text{namespace}())).$$

Node tests that occur explicitly in XPath queries must not use the types "root", "attribute", or "namespace"[5]. In XPath, axis applications χ and node tests t always come in *location step* expressions of the form χ::t. The node test n (where n is a node name or the wildcard *) is a shortcut for $\tau(n)$, where τ is the *principal node type* of χ. For the axis attribute, the principal node type is attribute, for namespace it is namespace, and for all other axes, it is element. For example, child::a is short for child::element(a) and child::* is short for child::element(*).

Note that for a set of nodes S and a typed axis χ, $\chi(S)$ can be computed in linear time – just as for the untyped axes of Section 3.

Let $<_{doc}$ be the binary document order relation, such that $x <_{doc} y$ (for two nodes $x, y \in$ dom) iff the opening tag of x precedes the opening tag of y in the (well-formed) document. The function $\text{first}_{<_{doc}}$ returns the first node in a set w.r.t. document order. We define the relation $<_{doc,\chi}$ relative to the axis χ as follows. For $\chi \in$ {self, child, descendant, descendant-or-self, following-sibling, following}, $<_{doc,\chi}$ is the standard document order relation $<_{doc}$. For the remaining axes, it is the reverse document order $>_{doc}$. Moreover, given a node x and a set of nodes S with $x \in S$, let $\text{idx}_\chi(x, S)$ be the index of x in S w.r.t. $<_{doc,\chi}$ (where 1 is the smallest index).

Given an XML Document Type Definition (DTD) [19] that uses the ID/IDREF feature, each element node of the document may be identified by a unique id. The function $\text{deref\_ids} : \text{string} \rightarrow 2^{\text{dom}}$ interprets its input string as a whitespace-separated list of keys and returns the set of nodes whose ids are contained in that list.

The function $\text{strval} : \text{dom} \rightarrow \text{string}$ returns the *string value* of a node, for the precise definition of which we refer to [18]. Notably, the string value of an element or root node x is the concatenation of the string values of descendant text nodes $\{y \mid \text{descendant}(\{x\}) \cap T(\text{text}())\}$ visited in document order. The functions to_string and to_number convert a number to a string resp. a string to a number according to the rules specified in [18].

This concludes our discussion of the XPath data model, which is complete except for some details related to namespaces. This topic is mostly orthogonal to our discussion, and extending our framework to also handle namespaces (without a penalty with respect to efficiency bounds) is an easy exercise. [6]

5 Semantics of XPath

In this section, we present a concise definition of the semantics of XPath 1 [18]. We assume the syntax of this language known, and cohere with its *unabbreviated* form [18]. This means that

- in all occurrences of the child or descendant axis in the XPath expression, the axis names have to be stated explicitly; for example, we write /descendant::a/child::b rather than //a/b.
- Bracketed condition expressions [e], where e is an expression that produces a number (see below), correspond to [position() = e] in unabbreviated syntax. For example, the abbreviated XPath expression //a[5], which refers to the fifth node (with respect to document order) occuring in the document which is labeled "a", is written as /descendant::a[position() = 5] in unabbreviated syntax.
- All type conversions have to be made explicit (using the conversion functions string, number, and boolean, which we will define below). For example, we write /descendant::a[boolean(child::b)] rather than /descendant::a[child::b].

Moreover, as XPath expression may use variables for which a given binding has to be supplied with the expression, each variable is replaced by the (constant) value of the input variable binding.

These assumptions do not cause any loss of generality, but reduce the number of cases we have to distinguish in the semantics definition below.

The main syntactic construct of XPath are *expressions*, which are of one of four types, namely *node set*, *number*, *string*, or *boolean*. Each expression evaluates relative to a context $\vec{c} = \langle x, k, n\rangle$ consisting of a *context node* x, a *context position* k, and a *context size* n [18]. By the *domain of contexts*, we mean the set

$$\mathbf{C} = \text{dom} \times \{\langle k, n\rangle \mid 1 \le k \le n \le |\text{dom}|\}.$$

Let

$$\begin{array}{rcl} \textit{ArithOp} & \in & \{+, -, *, \text{div}, \text{mod}\}, \\ \textit{RelOp} & \in & \{=, \neq, \le, <, \ge, >\}, \\ \textit{EqOp} & \in & \{=, \neq\}, \text{ and} \\ \textit{GtOp} & \in & \{\le, <, \ge, >\}. \end{array}$$

By slight abuse of notation, we identify these arithmetic and relational operations with their symbols in the remainder of this paper. However, it should be clear whether we refer to the operation or its symbol at any point. By $\pi, \pi_1, \pi_2, \ldots$ we denote location paths.

Definition 5.1 (Semantics of XPath) Each XPath expression returns a value of one of the following four types: number, node set, string, and boolean (abbreviated num, nset, str, and bool, respectively). Let $\mathcal{T}$ be an expression type and the semantics $[\![e]\!] : \mathbf{C} \rightarrow \mathcal{T}$ of XPath expression e be defined as follows.

$$\begin{array}{rcl} [\![\pi]\!](\langle x, k, n\rangle) & := & P[\![\pi]\!](x) \\ [\![\text{position}()]\!](\langle x, k, n\rangle) & := & k \\ [\![\text{last}()]\!](\langle x, k, n\rangle) & := & n \\ [\![\text{text}()]\!](\langle x, k, n\rangle) & := & \text{strval}(n) \end{array}$$

[5] These node tests are also redundant with '/' and the "attribute" and "namespace" axes.

[6] To be consistent, we also will not discuss the "local-name", "namespace-uri", and "name" core library functions [18].

Note that names used in node tests may be of the form NCName:*, which matches all names from a given namespace named NCNAME.

(* location paths relative to the root node *)
$P\llbracket/\pi\rrbracket(x) := P\llbracket\pi\rrbracket(\text{root})$

(* composition of location paths *)
$P\llbracket\pi_1/\pi_2\rrbracket(x) := \bigcup_{y \in P\llbracket\pi_1\rrbracket(x)} P\llbracket\pi_2\rrbracket(y)$

(* "disjunction" of location paths *)
$P\llbracket\pi_1|\pi_2\rrbracket(x) := P\llbracket\pi_1\rrbracket(x) \cup P\llbracket\pi_2\rrbracket(x)$

(* location steps *)
```
$P\llbracket\chi{::}t[e_1]\cdots[e_m]\rrbracket(x) :=$
begin
    $S := \{y \mid x\chi y,\ y \in T(t)\}$;
    for $1 \le i \le m$ (in ascending order) do
        $S := \{y \in S \mid \llbracket e_i\rrbracket(y, \text{idx}_\chi(y,S), |S|) = \text{true}\}$;
    return $S$;
end;
```

Figure 5: Standard semantics of location paths.

For all other kinds of expressions $e = Op(e_1, \ldots, e_m)$ mapping a context $\vec{c}$ to a value of type $\mathcal{T}$,

$$\llbracket Op(e_1, \ldots, e_m)\rrbracket(\vec{c}) := \mathcal{F}\llbracket Op\rrbracket(\llbracket e_1\rrbracket(\vec{c}), \ldots, \llbracket e_m\rrbracket(\vec{c})),$$

where $\mathcal{F}\llbracket Op\rrbracket : \mathcal{T}_1 \times \ldots \times \mathcal{T}_m \to \mathcal{T}$ is called the *effective semantics function* of *Op*. The function P is defined in Figure 5 and the effective semantics function $\mathcal{F}$ is defined in Table 2. □

To save space, we at times re-use function definitions in Table 2 to define others. However, our definitions are not circular and the indirections can be eliminated by a constant number of unfolding steps. Moreover, we define neither the number operations floor, ceiling, and round nor the string operations concat, starts-with, contains, substring-before, substring-after, substring (two versions), string-length, normalize-space, translate, and lang in Table 2, but it is very easy to obtain these definitions from the XPath 1 Recommendation [18].

The compatibility of our semantics definition (modulo the assumptions made in this paper to simplify the data model) with [18] can easily be verified by inspection of the latter document.

It is instructive to compare the definition of $P\llbracket\pi_1/\pi_2\rrbracket$ in Figure 5 with the procedure process-location-step of Section 2 and the claim regarding exponential-time query evaluation made there. In fact, if the semantics definition of [18] (or of this section, for that matter) is followed rigorously to obtain an analogous functional implementation, query evaluation using this implementation requires time exponential in the size of the queries.

6 Bottom-up Evaluation of XPath

In this section, we present a semantics and an algorithm for evaluating XPath queries in polynomial time which both use a "bottom-up" intuitition. We discuss

Expr. E **: Operator Signature**	**Semantics** $\mathcal{F}\llbracket E\rrbracket$
$\mathcal{F}\llbracket\text{constant number } v : \to \text{num}\rrbracket()$	v
$\mathcal{F}\llbracket \mathit{ArithOp} : \text{num} \times \text{num} \to \text{num}\rrbracket(v_1, v_2)$	$v_1\ \mathit{ArithOp}\ v_2$
$\mathcal{F}\llbracket\text{count} : \text{nset} \to \text{num}\rrbracket(S)$	$\lvert S\rvert$
$\mathcal{F}\llbracket\text{sum} : \text{nset} \to \text{num}\rrbracket(S)$	$\Sigma_{n \in S}$ to_number(strval(n))
$\mathcal{F}\llbracket\text{id} : \text{nset} \to \text{nset}\rrbracket(S)$	$\bigcup_{n \in S} \mathcal{F}\llbracket\text{id}\rrbracket(\text{strval}(n))$
$\mathcal{F}\llbracket\text{id} : \text{str} \to \text{nset}\rrbracket(s)$	deref_ids(s)
$\mathcal{F}\llbracket\text{constant string } s : \to \text{str}\rrbracket()$	s
$\mathcal{F}\llbracket\text{and} : \text{bool} \times \text{bool} \to \text{bool}\rrbracket(b_1, b_2)$	$b_1 \wedge b_2$
$\mathcal{F}\llbracket\text{or} : \text{bool} \times \text{bool} \to \text{bool}\rrbracket(b_1, b_2)$	$b_1 \vee b_2$
$\mathcal{F}\llbracket\text{not} : \text{bool} \to \text{bool}\rrbracket(b)$	$\neg b$
$\mathcal{F}\llbracket\text{true}() : \to \text{bool}\rrbracket()$	true
$\mathcal{F}\llbracket\text{false}() : \to \text{bool}\rrbracket()$	false
$\mathcal{F}\llbracket \mathit{RelOp} : \text{nset} \times \text{nset} \to \text{bool}\rrbracket(S_1, S_2)$	$\exists n_1 \in S_1, n_2 \in S_2 : \text{strval}(n_1)\ \mathit{RelOp}\ \text{strval}(n_2)$
$\mathcal{F}\llbracket \mathit{RelOp} : \text{nset} \times \text{num} \to \text{bool}\rrbracket(S, v)$	$\exists n \in S : \text{to\_number}(\text{strval}(n))\ \mathit{RelOp}\ v$
$\mathcal{F}\llbracket \mathit{RelOp} : \text{nset} \times \text{str} \to \text{bool}\rrbracket(S, s)$	$\exists n \in S : \text{strval}(n)\ \mathit{RelOp}\ s$
$\mathcal{F}\llbracket \mathit{RelOp} : \text{nset} \times \text{bool} \to \text{bool}\rrbracket(S, b)$	$\mathcal{F}\llbracket\text{boolean}\rrbracket(S)\ \mathit{RelOp}\ b$
$\mathcal{F}\llbracket \mathit{EqOp} : \text{bool} \times (\text{str} \cup \text{num} \cup \text{bool}) \to \text{bool}\rrbracket(b, x)$	$b\ \mathit{EqOp}\ \mathcal{F}\llbracket\text{boolean}\rrbracket(x)$
$\mathcal{F}\llbracket \mathit{EqOp} : \text{num} \times (\text{str} \cup \text{num}) \to \text{bool}\rrbracket(v, x)$	$v\ \mathit{EqOp}\ \mathcal{F}\llbracket\text{number}\rrbracket(x)$
$\mathcal{F}\llbracket \mathit{EqOp} : \text{str} \times \text{str} \to \text{bool}\rrbracket(s_1, s_2)$	$s_1\ \mathit{EqOp}\ s_2$
$\mathcal{F}\llbracket \mathit{GtOp} : (\text{str} \cup \text{num} \cup \text{bool}) \times (\text{str} \cup \text{num} \cup \text{bool}) \to \text{bool}\rrbracket(x_1, x_2)$	$\mathcal{F}\llbracket\text{number}\rrbracket(x_1)\ \mathit{GtOp}\ \mathcal{F}\llbracket\text{number}\rrbracket(x_2)$
$\mathcal{F}\llbracket\text{string} : \text{num} \to \text{str}\rrbracket(v)$	to_string(v)
$\mathcal{F}\llbracket\text{string} : \text{nset} \to \text{str}\rrbracket(S)$	**if** $S = \emptyset$ **then** "" **else** strval(first$_{<_{doc}}$(S))
$\mathcal{F}\llbracket\text{string} : \text{bool} \to \text{str}\rrbracket(b)$	**if** b=true **then** "true" **else** "false"
$\mathcal{F}\llbracket\text{boolean} : \text{str} \to \text{bool}\rrbracket(s)$	**if** $s \neq$ "" **then** true **else** false
$\mathcal{F}\llbracket\text{boolean} : \text{num} \to \text{bool}\rrbracket(v)$	**if** $v \neq \pm 0$ **and** $v \neq \mathit{NaN}$ **then** true **else** false
$\mathcal{F}\llbracket\text{boolean} : \text{nset} \to \text{bool}\rrbracket(S)$	**if** $S \neq \emptyset$ **then** true **else** false
$\mathcal{F}\llbracket\text{number} : \text{str} \to \text{num}\rrbracket(s)$	to_number(s)
$\mathcal{F}\llbracket\text{number} : \text{bool} \to \text{num}\rrbracket(b)$	**if** b=true **then** 1 **else** 0
$\mathcal{F}\llbracket\text{number} : \text{nset} \to \text{num}\rrbracket(S)$	$\mathcal{F}\llbracket\text{number}\rrbracket(\mathcal{F}\llbracket\text{string}\rrbracket(S))$

Table 2: XPath effective semantics functions.

Expression Type	Associated Relation R
num	$R \subseteq \mathbf{C} \times \mathbb{R}$
bool	$R \subseteq \mathbf{C} \times \{\text{true}, \text{false}\}$
nset	$R \subseteq \mathbf{C} \times 2^{\text{dom}}$
str	$R \subseteq \mathbf{C} \times \text{char}^*$

Table 3: Expression types and associated relations.

the intuitions which lead to polynomial time evaluation (which we call the "context-value table principle"), and establish the correctness and complexity results.

Definition 6.1 (Semantics) We represent the four XPath expression types nset, num, str, and bool using relations as shown in Table 3. The bottom-up semantics of expressions is defined via a semantics function

$$\mathcal{E}_\uparrow : \text{Expression} \to \text{nset} \cup \text{num} \cup \text{str} \cup \text{bool},$$

given in Table 4 and as

$$\mathcal{E}_\uparrow[\![Op(e_1, \ldots, e_m)]\!] := \{\langle \vec{c}, \mathcal{F}[\![Op]\!](v_1, \ldots, v_m)\rangle \mid \vec{c} \in \mathbf{C},\ \langle \vec{c}, v_1\rangle \in \mathcal{E}_\uparrow[\![e_1]\!],\ \ldots,\ \langle \vec{c}, v_m\rangle \in \mathcal{E}_\uparrow[\![e_m]\!]\}$$

for the remaining kinds of XPath expressions. □

Now, for each expression e and each $\langle x, k, n\rangle \in \mathbf{C}$, there is exactly one v s.t. $\langle x, k, n, v\rangle \in \mathcal{E}_\uparrow[\![e]\!]$, and which happens to be the value $[\![e]\!](\langle x, k, n\rangle)$ of e on $\langle x, k, n\rangle$ (see Definition 5.1).

Theorem 6.2 *Let e be an arbitrary XPath expression, $\langle x, k, n\rangle \in \mathbf{C}$ a context, and $v = [\![e]\!](\langle x, k, n\rangle)$ the value of e. Then, v is the unique value such that $\langle x, k, n, v\rangle \in \mathcal{E}_\uparrow[\![e]\!]$.*

The main principle that we propose at this point to obtain an XPath evaluation algorithm with polynomial-time complexity is the notion of a *context-value table* (i.e., a relation for each expression, as discussed above).

Context-value Table Principle. Given an expression e that occurs in the input query, the context-value table of e specifies all valid combinations of contexts $\vec{c}$ and values v, such that e evaluates to v in context $\vec{c}$. Such a table for expression e is obtained by first computing the context-value tables of the direct subexpressions of e and subsequently combining them into the context-value table for e. Given that the size of each of the context-value tables has a polynomial bound and each of the combination steps can be effected in polynomial time (all of which we can assure in the following), query evaluation in total under our principle also has a polynomial time bound[7]. □

Query Evaluation. The idea of Algorithm 6.3 below is so closely based on our semantics definition that its correctness follows directly from the correctness result of Theorem 6.2.

[7]The number of expressions to be considered is fixed with the parse tree of a given query.

Expr. E **: Operator Signature** **Semantics** $\mathcal{E}_\uparrow[\![E]\!]$
location step $\chi{::}t$: $\to$ nset $\{\langle x_0, k_0, n_0, \{x \mid x_0 \chi x,\ x \in T(t)\}\rangle \mid \langle x_0, k_0, n_0\rangle \in \mathbf{C}\}$
location step $E[e]$ over axis χ: nset $\times$ bool $\to$ nset $\{\langle x_0, k_0, n_0, \{x \in S \mid \langle x, \text{idx}_\chi(x, S), \lvert S\rvert, \text{true}\rangle \in \mathcal{E}_\uparrow[\![e]\!]\}\rangle \mid \langle x_0, k_0, n_0, S\rangle \in \mathcal{E}_\uparrow[\![E]\!]\}$
location path $/\pi$: nset $\to$ nset $\mathbf{C} \times \{S \mid \exists k, n : \langle \text{root}, k, n, S\rangle \in \mathcal{E}_\uparrow[\![\pi]\!]\}$
location path π_1/π_2 : nset $\times$ nset $\to$ nset $\{\langle x, k, n, z\rangle \mid 1 \le k \le n \le \lvert\text{dom}\rvert, \langle x, k_1, n_1, Y\rangle \in \mathcal{E}_\uparrow[\![\pi_1]\!], \bigcup_{y \in Y} \langle y, k_2, n_2, z\rangle \in \mathcal{E}_\uparrow[\![\pi_2]\!]\}$
location path $\pi_1 \mid \pi_2$: nset $\times$ nset $\to$ nset $\mathcal{E}_\uparrow[\![\pi_1]\!] \cup \mathcal{E}_\uparrow[\![\pi_2]\!]$
position() : $\to$ num $\{\langle x, k, n, k\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$
last() : $\to$ num $\{\langle x, k, n, n\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$
text() : $\to$ str $\{\langle x, k, n, \text{strval}(x)\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}$

Table 4: Expression relations for location paths, position(), last(), and text().

Algorithm 6.3 (Bottom-up algorithm for XPath)
Input: An XPath query Q;
Output: $\mathcal{E}_\uparrow[\![Q]\!]$.
Method:

```
let Tree($Q$) be the parse tree of query $Q$;
$\mathbf{R}$ := $\emptyset$; (* a set of context-value tables *)
for each atomic expression $l \in$ leaves(Tree($Q$)) do
    compute table $\mathcal{E}_\uparrow[\![l]\!]$ and add it to $\mathbf{R}$;
while $\mathcal{E}_\uparrow[\![\text{root(Tree}(Q))]\!] \notin \mathbf{R}$ do
begin
    take an Op$(l_1, \ldots, l_n) \in$ nodes(Tree(Q))
        s.t. $\mathcal{E}_\uparrow[\![l_1]\!], \ldots, \mathcal{E}_\uparrow[\![l_n]\!] \in \mathbf{R}$;
    compute $\mathcal{E}_\uparrow[\![\text{Op}(l_1, \ldots, l_n)]\!]$ using $\mathcal{E}_\uparrow[\![l_1]\!], \ldots, \mathcal{E}_\uparrow[\![l_n]\!]$;
    add $\mathcal{E}_\uparrow[\![\text{Op}(l_1, \ldots, l_n)]\!]$ to $\mathbf{R}$;
end;
return $\mathcal{E}_\uparrow[\![\text{root(Tree}(Q))]\!]$.
```

□

Example 6.4 Consider document $DOC(4)$ of Section 2. Let dom $= \{r, a, b_1, \ldots, b_4\}$, where r denotes the root node, a the document element node (the child of r, labeled a) and $b_1, \ldots, b_4$ denote the children of a in document order (labeled b). We want to evaluate the XPath query Q, which reads as

descendant::b/following-sibling::*[position() != last()]

over the input context $\langle a, 1, 1\rangle$. We illustrate how this evaluation can be done using Algorithm 6.3: First of all, we have to set up the parse tree

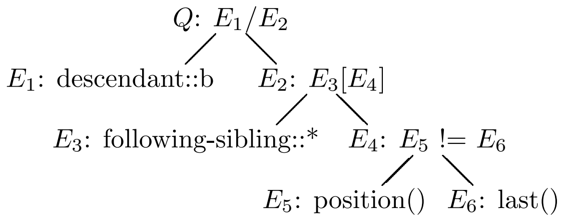
$\mathcal{E}_\uparrow[\![E_2]\!]$	
x	val
b_1	$\{b_2, b_3\}$
b_2	$\{b_3\}$

$\mathcal{E}_\uparrow[\![E_1]\!]$	
x	val
r	$\{b_1, b_2, b_3, b_4\}$
a	$\{b_1, b_2, b_3, b_4\}$

$\mathcal{E}_\uparrow[\![E_3]\!]$	
x	val
b_1	$\{b_2, b_3, b_4\}$
b_2	$\{b_3, b_4\}$
b_3	$\{b_4\}$

$\mathcal{E}_\uparrow[\![Q]\!]$	
x	val
r	$\{b_2, b_3\}$
a	$\{b_2, b_3\}$

Figure 6: Context-value tables of Example 6.4.



of Q with its 6 proper subexpressions $E_1, \ldots, E_6$. Then we compute the context-value tables of the leaf nodes E_1, E_3, E_5 and E_6 in the parse tree, and from the latter two the table for E_4. By combining E_3 and E_4, we obtain E_2, which is in turn needed for computing Q. The tables[8] for E_1, E_2, E_3 and Q are shown in Figure 6. Moreover,

$$\begin{aligned}
\mathcal{E}_\uparrow[\![E_5]\!] &= \{\langle x, k, n, k\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\} \\
\mathcal{E}_\uparrow[\![E_6]\!] &= \{\langle x, k, n, n\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\} \\
\mathcal{E}_\uparrow[\![E_4]\!] &= \{\langle x, k, n, k \neq n\rangle \mid \langle x, k, n\rangle \in \mathbf{C}\}
\end{aligned}$$

The most interesting step is the computation of $\mathcal{E}_\uparrow[\![E_2]\!]$ from the tables for E_3 and E_4. For instance, consider $\langle b_1, k, n, \{b_2, b_3, b_4\}\rangle \in \mathcal{E}_\uparrow[\![E_3]\!]$. b_2 is the first, b_3 the second, and b_4 the third of the three siblings following b_1. Thus, only for b_2 and b_3 is the condition E_2 (requiring that the position in set $\{b_2, b_3, b_4\}$ is different from the size of the set, three) satisfied. Thus, we obtain the tuple $\langle b_1, k, n, \{b_2, b_3\}\rangle$ which we add to $\mathcal{E}_\uparrow[\![E_2]\!]$.

We can read out the final result $\{b_2, b_3\}$ from the context-value table of Q. □

Theorem 6.5 *XPath can be evaluated bottom-up in polynomial time (combined complexity).*

Proof. Let $|Q|$ be the size of the query and $|D|$ be the size of the data. During the bottom-up computation of a query Q using Algorithm 6.3, $O(|Q|)$ relations ("context-value tables") are created. All relations have a functional dependency from the context (columns one to three) to the value (column four). The size of each relation is $O(|D|^3)$ times the maximum size of such values. The size of bool relations is bounded by $O(|D|^3)$ and the size of nset relations by $O(|D|^4)$.

[8]The k and n columns have been omitted. Full tables are obtained by computing the cartesian product of each table with $\{\langle k, n\rangle \mid 1 \leq k \leq n \leq |\text{dom}|\}$.

Numbers and strings computable in XPath are of size $O(|D| \cdot |Q|)$: "concat" on strings and arithmetic multiplication on numbers are the most costly operations (w.r.t. size increase of values) on strings and numbers there are[9]. Here, the lengths of the argument values add up such that we get to sizes $O(|D| \cdot |Q|)$ at worst, even in the relation representing the "top" expression Q itself.

The overall space bound of $O(|D|^4 \cdot |Q|^2)$ follows. Note that no significant additional amount of space is required for intermediate computations.

Let each context-value table be stored as a three-dimensional array, such that we can find the value for a given context $\langle x, k, n\rangle$ in constant time. Given m context-value tables representing expressions $e_1, \ldots, e_m$ and a context $\langle x, k, n\rangle$, any m-ary XPath operation $Op(e_1, \ldots, e_m)$ on context $\langle x, k, n\rangle$ can be evaluated in time $O(|D| \cdot I)$; again, I is the size of the input values and thus $O(|D| \cdot |Q|)$. This is not difficult to verify; it only takes very standard techniques to implement the XPath operations according to the definitions of Figure 2 (sometimes using auxiliary data structures created in a preprocessing step). The most costly operator is $RelOp$: nset $\times$ nset $\rightarrow$ bool, and this one also takes the most ingenuity. We assume a pre-computed table

$$\{\langle n_1, n_2\rangle \mid n_1, n_2 \in \text{dom},\ \text{strval}(n_1)\ RelOp\ \text{strval}(n_2)\}$$

with which we can carry out the operation in time $O(|D^2|)$ given two node sets.

It becomes clear that each of the expression relations can be computed in time $O(|D|^3 \cdot |D|^2 \cdot |Q|)$ at worst when the expression semantics tables of the direct subexpressions are given. (The $|Q|$ factor is due the size bound on strings and numbers generated during the computation.) Moreover, $O(|Q|)$ such computations are needed in total to evaluate Q. The $O(|D|^5 \cdot |Q|^2)$ time bound follows. □

Remark 6.6 Note that contexts can also be represented in terms of pairs of a current and a "previous" context node (rather than triples of a node, a position, and a size), which are defined relative to an axis and a node test (which, however, are fixed with the query). For instance, the corresponding ternary context for $\vec{c} = \langle x_0, x\rangle$ w.r.t. axis χ and node test t is $\langle x, \text{idx}_\chi(x, Y), |Y|\rangle$, where $Y = \{y \mid x_0 \chi y,\ y \in T(t)\}$. Thus, position and size values can be recovered on demand.

Moreover, it is possible to represent context-value tables of node set-typed expressions just as binary unnested relations (indeed, a construction that contains a previous node, a current node, and a node set as value would represent *two* location steps rather than

[9]For the conversion from a node set to a string or number, only the first node in the set is chosen. Of the string functions, only "concat" may produce a string longer than the input strings. The "translate" function of [18], for instance, does not allow for arbitrary but just single-character replacement, e.g. for case-conversion purposes.

```
(* location paths relative to the root node *)
$\mathcal{S}_\downarrow[\![/\pi]\!](X_1,\ldots,X_k) \;:=\; \mathcal{S}_\downarrow[\![\pi]\!](\underbrace{\{\text{root}\},\ldots,\{\text{root}\}}_{k \text{ times}})$

(* composition of location paths *)
$\mathcal{S}_\downarrow[\![\pi_1/\pi_2]\!](X_1,\ldots,X_k) := \mathcal{S}_\downarrow[\![\pi_2]\!](\mathcal{S}_\downarrow[\![\pi_1]\!](X_1,\ldots,X_k))$

(* "disjunction" of location paths *)
$\mathcal{S}_\downarrow[\![\pi_1 \mid \pi_2]\!](X_1,\ldots,X_k) :=$
    $\mathcal{S}_\downarrow[\![\pi_1]\!](X_1,\ldots,X_k) \cup^{\langle\rangle} \mathcal{S}_\downarrow[\![\pi_2]\!](X_1,\ldots,X_k)$

(* location steps *)
$\mathcal{S}_\downarrow[\![\chi::t[e_1]\cdots[e_m]]\!](X_1,\ldots,X_k) :=$
begin
    $S := \{\langle x,y\rangle \mid x \in \bigcup_{i=1}^k X_i,\ x\,\chi\,y, \text{ and } y \in T(t)\}$;
    for each $1 \le i \le m$ (in ascending order) do
    begin
        fix some order $\vec{S} = \langle\langle x_1,y_1\rangle,\ldots,\langle x_l,y_l\rangle\rangle$ for $S$;
        $\langle r_1,\ldots,r_l\rangle := \mathcal{E}_\downarrow[\![e_i]\!](t_1,\ldots,t_l)$
            where $t_j = \langle y_j,\ \mathrm{idx}_\chi(y_j,S_j),\ |S_j|\rangle$
            and $S_j := \{z \mid \langle x_j,z\rangle \in S\}$;
        $S := \{\langle x_i,y_i\rangle \mid r_i \text{ is true}\}$;
    end;
    for each $1 \le i \le k$ do
        $R_i := \{y \mid \langle x,y\rangle \in S, x \in X_i\}$;
    return $\langle R_1,\ldots,R_k\rangle$;
end;
```

Figure 7: Top-down evaluation of location paths.

one). Note that to move to this alternative form of representation, a number of changes in various aspects of our construction are necessary, which we do not describe here in detail.

Through these two changes, it is possible to obtain an improved worst-case time bound of $O(|D|^3 \cdot |Q|^2)$ for XPath query evaluation. □

7 Top-down Evaluation of XPath

In the previous section, we obtained a bottom-up semantics definition which led to a polynomial-time query evaluation algorithm for XPath. Despite this favorable complexity bound, this algorithm is still not practical, as usually many irrelevant intermediate results are computed to fill the context-value tables which are not used later on. Next, building on the context-value table principle of Section 6, we develop a top-down algorithm based on vector computation for which the favorable (worst-case) complexity bound carries over but in which the computation of a large number of irrelevant results is avoided.

Given an m-ary operation $Op : D^m \to D$, its vectorized version $Op^{\langle\rangle} : (D^k)^m \to D^k$ is defined as

$$Op^{\langle\rangle}(\langle x_{1,1},\ldots,x_{1,k}\rangle,\ldots,\langle x_{m,1},\ldots,x_{m,k}\rangle) := \langle Op(x_{1,1},\ldots,x_{m,1}),\ldots,Op(x_{1,k},\ldots,x_{m,k})\rangle$$

For instance, $\langle X_1,\ldots,X_k\rangle \cup^{\langle\rangle} \langle Y_1,\ldots,Y_k\rangle := \langle X_1\cup Y_1,\ldots,X_k\cup Y_k\rangle$. Let

$$\mathcal{S}_\downarrow : \text{LocationPath} \to \text{List}(2^{\text{dom}}) \to \text{List}(2^{\text{dom}})$$

be the auxiliary semantics function for location paths defined in Figure 7. We basically distinguish the same cases (related to location paths) as for the bottom-up semantics $\mathcal{E}_\uparrow[\![\pi]\!]$. Given a location path π and a list $\langle X_1,\ldots,X_k\rangle$ of node sets, $\mathcal{S}_\downarrow$ determines a list $\langle Y_1,\ldots,Y_k\rangle$ of node sets, s.t. for every $i \in \{1,\ldots,k\}$, the nodes reachable from the context nodes in X_i via the location path π are precisely the nodes in Y_i. $\mathcal{S}_\downarrow[\![\pi]\!]$ can be obtained from the relations $\mathcal{E}_\uparrow[\![\pi]\!]$ as follows. A node y is in Y_i iff there is an $x \in X_i$ and some p, s such that $\langle x,p,s,y\rangle \in \mathcal{E}_\uparrow[\![\pi]\!]$.

Definition 7.1 The semantics function $\mathcal{E}_\downarrow$ for arbitrary XPath expressions is of the following type:

$$\mathcal{E}_\downarrow : \text{XPathExpression} \to \text{List}(\mathbf{C}) \to \text{List}(\text{XPathType})$$

Given an XPath expression e and a list $(\vec{c}_1,\ldots,\vec{c}_l)$ of contexts, $\mathcal{E}_\downarrow$ determines a list $\langle r_1,\ldots,r_l\rangle$ of results of one of the XPath types number, string, boolean, or node set. $\mathcal{E}_\downarrow$ is defined as

$\mathcal{E}_\downarrow[\![\pi]\!](\langle x_1,k_1,n_1\rangle,\ \ldots,\ \langle x_l,k_l,n_l\rangle) :=$
$\quad\mathcal{S}_\downarrow[\![\pi]\!](\{x_1\},\ldots,\{x_l\})$

$\mathcal{E}_\downarrow[\![\text{position()}]\!](\langle x_1,k_1,n_1\rangle,\ \ldots,\ \langle x_l,k_l,n_l\rangle) :=$
$\quad\langle k_1,\ldots,k_l\rangle$

$\mathcal{E}_\downarrow[\![\text{last()}]\!](\langle x_1,k_1,n_1\rangle,\ \ldots,\ \langle x_l,k_l,n_l\rangle) :=$
$\quad\langle n_1,\ldots,n_l\rangle$

$\mathcal{E}_\downarrow[\![\text{text()}]\!](\langle x_1,k_1,n_1\rangle,\ \ldots,\ \langle x_l,k_l,n_l\rangle) :=$
$\quad\langle \text{strval}(x_1),\ldots,\text{strval}(n_l)\rangle$

and

$\mathcal{E}_\downarrow[\![Op(e_1,\ldots,e_m)]\!](\vec{c}_1,\ldots,\vec{c}_l) :=$
$\quad\mathcal{F}[\![Op]\!]^{\langle\rangle}(\mathcal{E}_\downarrow[\![e_1]\!](\vec{c}_1,\ldots,\vec{c}_l),\ldots,\mathcal{E}_\downarrow[\![e_m]\!](\vec{c}_1,\ldots,\vec{c}_l))$

for the remaining kinds of expressions. □

Example 7.2 Consider the XPath query

/descendant::a[count(descendant::b/child::c)
+ position() < last()]/child::d

Let $L = \langle\langle y_1,1,l\rangle,\ldots,\langle y_l,l,l\rangle\rangle$, where the y_i are those nodes reachable from the root node through the descendant axis and which are labeled "a". The query is evaluated top-down as

$$\mathcal{S}_\downarrow[\![\text{child::d}]\!](\mathcal{S}_\downarrow[\![\text{descendant::a}[e]]\!](\{\mathit{root}\}))$$

where $\mathcal{E}_\downarrow[\![e]\!](L)$ is computed as

$$\mathcal{F}[\![\text{count}]\!]^{\langle\rangle}(\pi) +^{\langle\rangle} \mathcal{E}_\downarrow[\![\text{position()}]\!](L) <^{\langle\rangle} \mathcal{E}_\downarrow[\![\text{last()}]\!](L)$$

and

$$\pi = \mathcal{S}_\downarrow[\![\text{child::c}]\!](\mathcal{S}_\downarrow[\![\text{descendant::b}]\!](\{y_1\},\ldots,\{y_l\})).$$

Note that the arity of the tuples used to compute the outermost location path is one, while it is l for e. □

Example 7.3 Given the query Q, data, and context $\langle a, 1, 1\rangle$ of Example 6.4, we evaluate Q as $\mathcal{E}_\downarrow[\![Q]\!](\langle a,1,1\rangle) = \mathcal{S}_\downarrow[\![E_2]\!](\mathcal{S}_\downarrow[\![\text{descendant::b}]\!](\{a\}))$. Again, E_2 is the subexpression

following-sibling::*[position() != last()].

First, we obtain $\mathcal{S}_\downarrow[\![\text{descendant::b}]\!](\{a\}) = \langle\{b_1, b_2, b_3, b_4\}\rangle$. To compute the location step $\mathcal{S}_\downarrow[\![E_2]\!](\langle\{b_1, b_2, b_3, b_4\}\rangle)$, we proceed as described in the algorithm of Figure 7. We initially obtain the set

$$S = \{\langle b_1, b_2\rangle, \langle b_1, b_3\rangle, \langle b_1, b_4\rangle, \langle b_2, b_3\rangle, \langle b_2, b_4\rangle, \langle b_3, b_4\rangle\}$$

and the list of contexts $\vec{t} = \langle\langle b_2, 1, 3\rangle, \langle b_3, 2, 3\rangle, \langle b_4, 3, 3\rangle, \langle b_3, 1, 2\rangle, \langle b_4, 2, 2\rangle, \langle b_4, 1, 1\rangle\rangle$.

The check of condition E_4 returns the filter

$$\vec{r} = \langle \text{true}, \text{true}, \text{false}, \text{true}, \text{false}, \text{false}\rangle.$$

which is applied to S to obtain

$$S = \{\langle b_1, b_2\rangle, \langle b_1, b_3\rangle, \langle b_2, b_3\rangle\}$$

Thus, the query returns $\langle\{b_2, b_3\}\rangle$. □

The correctness of the top-down semantics follows immediately from the corresponding result in the bottom-up case and from the definition of $\mathcal{S}_\downarrow$ and $\mathcal{E}_\downarrow$.

Theorem 7.4 *(Correctness of $\mathcal{E}_\downarrow$) Let e be an arbitrary XPath expression. Then,*

$$\langle v_1, \dots, v_l\rangle = \mathcal{E}_\downarrow[\![e]\!](\vec{c}_1, \dots, \vec{c}_l)$$

iff

$$\langle \vec{c}_1, v_1\rangle, \dots, \langle \vec{c}_l, v_l\rangle \in \mathcal{E}_\uparrow[\![e]\!].$$

$\mathcal{S}_\downarrow$ and $\mathcal{E}_\downarrow$ can be immediately transformed into function definitions in a top-down algorithm. We thus have to define one evaluation function for each case of the definition of $\mathcal{S}_\downarrow$ and $\mathcal{E}_\downarrow$, respectively. The functions corresponding to the various cases of $\mathcal{S}_\downarrow$ have a location path and a list of node sets of variable length $(X_1, \dots, X_k)$ as input parameter and return a list $(R_1, \dots, R_k)$ of node sets of the same length as result. Likewise, the functions corresponding to $\mathcal{E}_\downarrow$ take an arbitrary XPath expression and a list of contexts as input and return a list of XPath values (which can be of type num, str, bool or nset). Moreover, the recursions in the definition of $\mathcal{S}_\downarrow$ and $\mathcal{E}_\downarrow$ correspond to recursive function calls of the respective evaluation functions. Analogously to Theorem 6.5, we get

Theorem 7.5 *The immediate functional implementation of $\mathcal{E}_\downarrow$ evaluates XPath queries in polynomial time (combined complexity).*

Finally, note that using arguments relating the top-down method of this section with (join) optimization techniques in relational databases, one may argue that the context-value table principle is also the basis of the polynomial-time bound of Theorem 7.5.

8 Linear-time Fragments of XPath

8.1 Core XPath

In this section, we define a fragment of XPath (called Core XPath) which constitutes a clean logical core of XPath (cf. [6, 7]). The only objects that are manipulated in this language are sets of nodes (i.e., there are no arithmetical or string operations). Besides from these restrictions, the full power of location paths is supported, and so is the matching of such paths in condition predicates (with an "exists" semantics), and the closure of such condition expressions with respect to boolean operations "and", "or", and "not".

We define a mapping of each query in this language to a simple algebra over the set operations $\cap$, $\cup$, '$-$', χ (the axis functions from Definition 3.1), and an operation $\frac{\text{dom}}{\text{root}}(S) := \{x \in \text{dom} \mid \text{root} \in S\}$, i.e. $\frac{\text{dom}}{\text{root}}(S)$ is dom if root $\in S$ and $\emptyset$ otherwise.

Note that each XPath axis has a natural *inverse*: self^{-1} = self, child^{-1} = parent, descendant^{-1} = ancestor, descendant-or-self^{-1} = ancestor-or-self, following^{-1} = preceding, and following-sibling^{-1} = preceding-sibling.

Lemma 8.1 *Let χ be an axis. For each pair of nodes $x, y \in$ dom, $x\chi y$ iff $y\chi^{-1}x$.*

(Proof by a very easy induction.)

Definition 8.2 Let the (abstract) syntax of the Core XPath language be defined by the EBNF grammar

cxp:	locationpath | '/' locationpath
locationpath:	locationstep ('/' locationstep)*
locationstep:	χ '::' t | χ '::' t '[' pred ']'
pred:	pred 'and' pred | pred 'or' pred
	| 'not' '(' pred ')' | cxp | '(' pred ')'

"cxp" is the start production, χ stands for an axis (see above), and t for a "node test" (either an XML tag or "*", meaning "any label"). The semantics of Core XPath queries is defined by a function $\mathcal{S}_\rightarrow$

$$\begin{array}{rcl}
\mathcal{S}_\rightarrow[\![\chi{::}t[e]]\!](N_0) & := & \chi(N_0) \cap T(t) \cap \mathcal{E}_1[\![e]\!] \\
\mathcal{S}_\rightarrow[\![/\chi{::}t[e]]\!](N_0) & := & \chi(\{\text{root}\}) \cap T(t) \cap \mathcal{E}_1[\![e]\!] \\
\mathcal{S}_\rightarrow[\![\pi/\chi{::}t[e]]\!](N_0) & := & \chi(\mathcal{S}_\rightarrow[\![\pi]\!](N_0)) \cap T(t) \cap \mathcal{E}_1[\![e]\!] \\
\mathcal{S}_\leftarrow[\![\chi{::}t[e]]\!] & := & \chi^{-1}(T(t) \cap \mathcal{E}_1[\![e]\!]) \\
\mathcal{S}_\leftarrow[\![\chi{::}t[e]/\pi]\!] & := & \chi^{-1}(\mathcal{S}_\leftarrow[\![\pi]\!] \cap T(t) \cap \mathcal{E}_1[\![e]\!]) \\
\mathcal{S}_\leftarrow[\![/\pi]\!] & := & \dfrac{\text{dom}}{\text{root}}(\mathcal{S}_\leftarrow[\![\pi]\!]) \\
\mathcal{E}_1[\![e_1 \text{ and } e_2]\!] & := & \mathcal{E}_1[\![e_1]\!] \cap \mathcal{E}_1[\![e_2]\!] \\
\mathcal{E}_1[\![e_1 \text{ or } e_2]\!] & := & \mathcal{E}_1[\![e_1]\!] \cup \mathcal{E}_1[\![e_2]\!] \\
\mathcal{E}_1[\![\text{not}(e)]\!] & := & \text{dom} - \mathcal{E}_1[\![e]\!] \\
\mathcal{E}_1[\![\pi]\!] & := & \mathcal{S}_\leftarrow[\![\pi]\!]
\end{array}$$

where N_0 is a set of context nodes or dom and a query π evaluates as $\mathcal{S}_\rightarrow[\![\pi]\!](N_0)$. □

Example 8.3 The Core XPath query

/descendant::a/child::b[child::c/child::d or not(following::*)]

is evaluated as specified by the query tree

$$\begin{tikzcd}
 & & & & {} & & & & \\
 & & & & \cap \arrow[u] & & & & \\
 & & \cap \arrow[urr] & & & & \cup \arrow[ull] & & \\
 & \text{child} \arrow[ur] & & T(b) \arrow[ul] & & \text{parent} \arrow[ur] & & - \arrow[ul] & \\
 & \cap \arrow[u] & & & & \cap \arrow[u] & \text{dom} \arrow[ur] & & \text{preceding} \arrow[ul] \\
\text{descendant} \arrow[ur] & & T(a) \arrow[ul] & & T(c) \arrow[ur] & & \text{parent} \arrow[ul] & & \text{dom} \arrow[u] \\
\{\text{root}\} \arrow[u] & & & & & & T(d) \arrow[u] & &
\end{tikzcd}$$

(There are alternative but equivalent query trees due to the associativity and commutativity of some of our operators.) □

The semantics of XPath and Core XPath (defined using $\mathcal{S}_{\leftarrow}$, $\mathcal{S}_{\rightarrow}$, and $\mathcal{E}_1$) coincide in the following way:

Theorem 8.4 *Let π be a Core XPath query and $N_0 \subseteq$ dom be a set of context nodes. Then,*

$$\mathcal{S}_{\leftarrow}[\![\pi]\!] = \{x \mid \mathcal{S}_{\downarrow}[\![\pi]\!](\{x\}) \neq \emptyset\}$$
$$\mathcal{E}_1[\![e]\!] = \{x \mid \mathcal{E}_{\downarrow}[\![e]\!](\{\langle x, 1, 1\rangle\})\}$$
$$\langle \mathcal{S}_{\rightarrow}[\![\pi]\!](N_0)\rangle = \mathcal{S}_{\downarrow}[\![\pi]\!](\langle N_0\rangle).$$

This can be shown by easy induction proofs. Thus, Core XPath (evaluated using $\mathcal{S}_{\rightarrow}$) is a fragment of XPath, both syntactically and semantically.

Theorem 8.5 *Core XPath queries can be evaluated in time $O(|D| * |Q|)$, where $|D|$ is the size of the data and $|Q|$ is the size of the query.*

Proof Given a query Q, it can be rewritten into an algebraic expression E over the operations χ, $\cup$, $\cap$, '$-$', and $\frac{\text{dom}}{\text{root}}$ using $\mathcal{S}_{\rightarrow}$, $\mathcal{S}_{\leftarrow}$, and $\mathcal{E}_1$ in time $O(|Q|)$. Each of the operations in our algebra can be carried out in time $O(|D|)$. Since at most $O(|Q|)$ such operations need to be carried out to process E, the complexity bound follows. □

8.2 XPatterns

We extend our linear-time fragment Core XPath by the operation id: nset → nset of Table 4 by defining "id" as an axis relation

$$id := \{\langle x_0, x\rangle \mid x_0 \in \text{dom},\ x \in \text{deref\_ids}(\text{strval}(x_0))\}$$

Queries of the form $\pi_1/\text{id}(\pi_2)/\pi_3$ are now treated as $\pi_1/\pi_2/\text{id}/\pi_3$.

Lemma 8.6 Let $\pi_1/\text{id}(\pi_2)/\pi_3$ be an XPath query s.t. $\pi_1/\pi_2/\text{id}/\pi_3$ is a query in Core XPath with the "id" axis. Then, the semantics of the two queries relative to a set of context nodes $N_0 \in$ dom coincide, $\mathcal{S}_{\downarrow}[\![\pi_1/\text{id}(\pi_2)/\pi_3]\!](\langle N_0\rangle) = \mathcal{S}_{\rightarrow}[\![\pi_1/\pi_2/\text{id}/\pi_3]\!](N_0)$. □

Theorem 8.7 *Queries in Core XPath with the "id" axis can be evaluated in time $O(|D| * |Q|)$.*

Proof. The interesting part of this proof is to define a function id: $2^{\text{dom}} \rightarrow 2^{\text{dom}}$ and its inverse consistent with the functions of Definition 3.1 which is computable in linear time. We make use of a binary auxiliary relation "ref" which contains a tuple of nodes $\langle x, y\rangle$ iff the text belonging to x in the XML document, but which is directly inside it and not further down in any of its descendants, contains a whitespace-separated string referencing the identifier of node y.

Example. Let $\text{id}(i) = n_i$. For the XML document ⟨t id=1⟩ 3 ⟨t id=2⟩ 1 ⟨/t⟩ ⟨t id=3⟩ 1 2 ⟨/t⟩ ⟨/t⟩, we have ref $:= \{\langle n_1, n_3\rangle,\ \langle n_2, n_1\rangle,\ \langle n_3, n_1\rangle,\ \langle n_3, n_2\rangle\}$. □

"ref" can be efficiently computed in a preprocessing step. It does not satisfy any functional dependencies, but it is guaranteed to be of linear size w.r.t. the input data (however, not in the tree nodes). Now we can encode id(S) as those nodes reachable from S *and its descendants* using "ref".

$$\text{id}(S) := \{y \mid x \in \text{descendant-or-self}(S),\ \langle x, y\rangle \in \text{ref}\}$$
$$\text{id}^{-1}(S) := \text{ancestor-or-self}(\{x \mid \langle x, y\rangle \in \text{ref},\ y \in S\})$$

This computation can be performed in linear time. □

We may define XPatterns as the smallest language that subsumes Core XPath and the XSLT Pattern language of [17] (see also [14] for a good and formal overview of this language) and is (syntactically) contained in XPath. Stated differently, it is obtained by extending the language of [17] without the first-of-type and last-of-type predicates (which do not exist in XPath) to support all of the XPath axes. As pointed out in the introduction, XPatterns is an interesting and practically useful query language. Surprisingly, XPatterns queries can be evaluated in linear time.

Theorem 8.8 *Let D be an XML document and Q be an XPatterns query. Then, Q can be evaluated on D in time $O(|D| * |Q|)$.*

Proof (Rough Sketch). XPatterns extends Core XPath by the "id" axis and a number of features which are definable as unary predicates, of which we give an overview in Table 5. It becomes clear by considering the semantics definition of [14] that after parsing the query, one knows of a fixed number of predicates to populate, and this action takes time $O(|D|)$ for each. Thus, since this computation precedes the query evaluation – which has a time bound of $O(|D| * |Q|)$ – this does not pose a problem. "id(s)" (for some fixed string s) may only occur at the beginning of a path, thus in a query of the form id(s)/π, π is evaluated relative to the set id(s) just as, say, {root} is for query /π. □

"@n", "@*", "text()", "comment()", "pi(n)", and "pi()" (where n is a label) are simply sets provided with the document (similar to those obtained through the node test function T).
"=s" (s is a string) can be encoded as a unary predicate whose extension can be computed using string search in the document before the evaluation of our query starts. Clearly, this can be done in linear time.
first-of-any := $\{y \in \text{dom} \mid \nexists x : \text{nextsibling}(x, y)\}$ last-of-any := $\{x \in \text{dom} \mid \nexists y : \text{nextsibling}(x, y)\}$
"id(s)" is a unary predicate and can easily be computed (in linear time) before the query evaluation.

Table 5: Some unary predicates of XLST Patterns [17].

$\|Q\|$	IE6 10	IE6 20	IE6 200	New 10	New 20	New 200
1				0	0	0.02
2			2	0	0	0.05
3			346	0	0	0.06
4		1	-	0	0	0.07
5		21	-	0	0	0.10
6	5	406	-	0	0.01	0.11
7	42	-	-	0.01	0.01	0.13
8	437	-	-	0	0.01	0.16
⋮						
16	-	-	-	0.01	0.02	0.30

Figure 8: Benchmark results in seconds for IE6 vs. our implementation ("New"), on the queries of Experiment 2 and document sizes 10, 20, and 200.

Let Σ be a finite set of all possible node names that a document may use (e.g., given through a DTD). Note that the unary first-of-type and last-of-type predicates can be computed in time $O(|D| * |\Sigma|)$ when parsing the document, but are of size $O(|D|)$:

$$\text{first-of-type()} := \bigcup_{l \in \Sigma} \big(T(l) - \text{nextsibling}^{+}(T(l))\big)$$

$$\text{last-of-type()} := \bigcup_{l \in \Sigma} \big(T(l) - (\text{nextsibling}^{-1})^{+}(T(l))\big)$$

where $R^{+} = R.R^{*}$.

9 Conclusions

In this paper, we presented the first XPath query evaluation algorithm that runs in polynomial time with respect to the size of both the data and of the query. Our results will empower XPath engines to be able to deal efficiently with very sophisticated queries.

We have made a main-memory implementation of the top-down algorithm of Section 7. Figure 8 compares it to IE6 along the assumptions made in Experiment 2 (i.e., the queries of which were strictly the most demanding of all three experiments). It shows that our algorithm scales linearly in the size of the queries and quadratically (for this class of queries) in the size of the data. Our implementation is still an early, naive prototype without any optimizations, and which strictly coheres to the specification given in this paper. We plan to significantly improve on its real-world runtime in terms of data in the future. Resources and further benchmarks that become available in the course of this effort will be made accessible at

`http://www.xmltaskforce.com`

Note that work subsequent to this [8] discusses further large XPath fragments which can be processed in improved time and space bounds. In the future, we intend to work on algorithms for processing XPath with disk access and with streaming XML data.

Acknowledgments

We thank G. Moerkotte and the anonymous reviewers of VLDB 2002, whose constructive comments have helped to considerably improve this paper, and P. Fankhauser for a wealth of pointers to the literature.

We also thank J. Siméon for pointing out to us that the mapping from XPath to the XML Query Algebra, which will be the preferred semantics definition for XPath 2, in an direct functional implementation also leads to exponential-time query processing on XPath 1 (which is a fragment of XPath 2).

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan Kaufmann Publishers, 2000.

[2] G. J. Bex, S. Maneth, and F. Neven. A Formal Model for an Expressive Fragment of XSLT. In *Proc. CL 2000*, LNCS 1861, pages 1137–1151. Springer, 2000.

[3] N. Bruno, D. Srivastava, and N. Koudas. "Holistic Twig Joins: Optimal XML Pattern Matching". In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, Madison, Wisconsin, June 2002.

[4] J. Clark. XT. A Java Implementation of XSLT http://www.jclark.com/xml/xt.html/.

[5] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath. In *Proc. KRDB 2001*, CEUR Workshop Proceedings 45, 2001.

[6] G. Gottlob and C. Koch. "Monadic Datalog and the Expressive Power of Web Information Extraction Languages". In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'02)*, pages 17–28, Madison, Wisconsin, 2002.

[7] G. Gottlob and C. Koch. “Monadic Queries over Tree-Structured Data”. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 189–202, Copenhagen, Denmark, July 2002.

[8] G. Gottlob, C. Koch, and R. Pichler. “XPath Query Evaluation: Improving Time and Space Efficiency”. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE’03)*, Bangalore, India, Mar. 2003. to appear.

[9] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, Department of Computer Science, University of Helsinki, Nov. 1992. Report A-1992-6.

[10] G. Miklau and D. Suciu. “Containment and Equivalence for an XPath Fragment”. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’02)*, pages 65–76, Madison, Wisconsin, 2002.

[11] T. Milo, D. Suciu, and V. Vianu. “Typechecking for XML Transformers”. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’00)*, pages 11–22, 2000.

[12] D. Shasha, J. T. L. Wang, and R. Giugno. “Algorithmics and Applications of Tree and Graph Searching”. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’02)*, June 3 – 5 2002.

[13] P. Wadler. “Two Semantics for XPath”, 2000. Draft paper available at http://www.research.avayalabs.com/user/wadler/.

[14] P. Wadler. “A Formal Semantics of Patterns in XSLT”. In *Markup Technologies*, Philadelphia, December 1999. Revised version in Markup Languages, MIT Press, June 2001.

[15] P. T. Wood. “On the Equivalence of XML Patterns”. In *Proc. 1st International Conference on Computational Logic (CL 2000)*, LNCS 1861, pages 1152–1166, London, UK, July 2000. Springer-Verlag.

[16] World Wide Web Consortium. DOM Specification http://www.w3c.org/DOM/.

[17] World Wide Web Consortium. XSL Working Draft http://www.w3.org/TR/1998/WD-xsl-19981216.

[18] World Wide Web Consortium. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/, Nov. 1999.

[19] World Wide Web Consortium. “Extensible Markup Language (XML) 1.0 (Second Edition)”, Oct. 2000. http://www.w3.org/TR/REC-xml.

[20] World Wide Web Consortium. “XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft (Aug. 16th 2002), 2002. `http://www.w3.org/TR/query-algebra/`.

[21] Xalan-Java version 2.2.D11. http://xml.apache.org/xalan-j/.