XPath Leashed

Michael Benedikt
Oxford University Computing Laboratory
michael.benedikt@comlab.ox.ac.uk
Christoph Koch
Cornell University
koch@cs.cornell.edu

This survey gives an overview of formal results on the XML query language XPath. We identify several important fragments of XPath, focusing on subsets of XPath 1.0. We then give results on the expressiveness of XPath and its fragments compared to other formalisms for querying trees, algorithms and complexity bounds for evaluation of XPath queries, and static analysis of XPath queries.

Categories and Subject Descriptors: H.2.3 [**Languages**]: Query languages

1. INTRODUCTION

XPath [World Wide Web Consortium 1999a] is a language for matching paths and, more generally, patterns in tree-structured data and XML documents. These patterns may use either just purely the tree structure of an XML document or data values occurring in the document as well.

XPath is used as a component in XML query languages (in particular, XQuery [World Wide Web Consortium 2002] and XSLT [World Wide Web Consortium 1999b]), specifications (e.g., XML Schema [World Wide Web Consortium 2001]), update languages (e.g., [Sur et al. 2004]), subscription systems (e.g., [Altinel and Franklin 2000; Chan et al. 2000]) and XML access control (e.g., [Fan et al. 2004]). Because XPath is ubiquitous in programming tools for manipulating XML documents, and XPath processing is a key component of these tools, hundreds if not thousands of papers have appeared over the years dealing with the evaluation and analysis of XPath. Indeed the popularity of XPath as a formalism may be a factor in the explosive growth of XML, as well as an effect.

The XPath standard has its rough edges, but there is an essential navigational core that is an elegant *modal language*. In this core of XPath there is no explicit notion of variable, and modal step expressions allow for navigation relative to a context node and thus can only "see" one element of the document at a time.

An important property of XPath (which follows from its syntactic restrictions that make it a modal language) is that fragments correspond to certain bounded-variable logics. From these logics, XPath inherits nice graph-theoretic properties on the "dependency graphs" of its queries. In particular, the queries have bounded tree-width and bounded hypertree-width. These properties render them amenable to efficient evaluation [Gottlob et al. 2005]. XPath is quite unique in the sense that (1) it is a widely used practical language that naturally obeys syntactic restrictions that lead to bounded (hyper)tree-width *and* (2) bounded (hyper)tree-width is of immediate practical relevance to efficient evaluation. (1) is true for modal languages used in verification, but (2) is not, as the query evaluation techniques used in the context of those languages are quite different [Burch et al. 1990; Clarke et al. 2000].

In this survey, we present the main fundamental results regarding XPath that have been developed since its introduction. These results can be grouped into the categories *expressiveness*, *complexity*, and *static analysis* of XPath.

—We give a detailed account of the known expressiveness results for XPath, but also give a number of new results. In particular, we review the connections between XPath and first-order logic. The main results are that there are first-order queries not expressible in navigational XPath, but that navigational XPath expresses precisely the two-variable first-order queries over the navigational structure of XML documents. We show that the navigational XPath fragment extended by the aggregation features of XPath does express all first-order queries. We also survey characterizations of fragments of XPath in terms of tree-pattern queries, and characterize XPath in terms of automata.

—We present an in-depth study of XPath complexity and efficient evaluation that revolves around graph-theoretic properties of XPath queries. Large portions of the XPath language can be processed by algorithms that can work in parallel or in streaming fashion. These issues have been studied extensively in the literature, but we present an overview here as well.

—We also survey static analysis problems for XPath, in particular the satisfiability and the containment problem. These have diverse applications such as in the context of XML query optimization, maintaining integrity, and answering queries using views.

The structure of this article is as follows. In Section 2, we present the data model and XPath fragments considered in this article, and give their semantics. Section 3 studies the expressive power of our XPath fragments, relating them to various logics, and the cost (and blow-up) of translating between such languages. Section 4 discusses the main results on the complexity of XPath and of efficient query evaluation, addressing efficient algorithms both in a classical and a stream processing framework, as well as lower bounds. Finally, Section 5 surveys the state of the art of research on static analysis problems for XPath.

For the central results in this survey, proofs are given. In some cases, we give proofs that are simplifications of those in the literature, while in other cases we give new proofs.

2. FRAMEWORK

Any fundamental research study of XPath has to decide what XPath really is – that is, to distinguish which language features of many to focus on. XPath officially refers to the World Wide Web Consortium's (W3C) standard language. This is a moving target, and indeed while virtually all research on XPath has focussed on the XPath 1.0 standard [World Wide Web Consortium 1999a], there is an extension, XPath 2.0 [World Wide Web Consortium 2007], which has recently reached Recommendation status.

Thus the first task for a formal study is to isolate a particular subset of the language with attractive properties, and to distinguish essential language features from provisional design decisions. In this survey we focus exclusively on XPath 1.0, and take the modal and step primitives that characterize XPath 1.0 as the definitive features of the language. Furthermore, since XPath 1.0 is still a large language, we concentrate on a sublanguage that exhibits the basic navigation and data manipulation features. The principal aspects that we ignore are string-manipulation, type conversions, and construction of string values from document fragments. For the most part the operations available at the value level do not affect our basic results, but we will comment briefly on their impact in the appropriate sections. The largest language we consider, denoted OrdXPath, allows for the selection of nodes based on navigation within the tree structure, data value comparisons, aggregation, and node position arithmetic. Within OrdXPath, we will delineate a hierarchy of sublanguages of XPath 1.0 to which more precise expressiveness or complexity bounds apply. We will refer to these sublanguages as XPath fragments. Of particular interest will be Navigational XPath (NavXPath), which deals only with the underlying tree structure of the document. All the fragments considered in this survey are formally introduced in Section 2.2.

The languages of this survey can thus be thought of as subsets of XPath 1.0 capturing the more important features of the language. In our definition of NavXPath, we make some small superficial departures from the concrete syntax of XPath 1.0. We do this because clean syntax in some cases allows for more readable proofs. We discuss these deviations from standard syntax in the text.

2.1 Data Model

A *signature* (or *vocabulary*) is a set of relation and function names. A *relational signature* is one consisting only of relation names (i.e., a relational schema). A *σ-structure* is a structure (or database) of signature σ. As a convention, given a structure $\mathcal{A}$, we use A (the name of the structure set in roman font) to denote its domain and $|\mathcal{A}|$ to denote the size of the structure in a reasonable machine-representation (cf. e.g. [Immerman 1999; Libkin 2004]).

Let Σ be a finite alphabet of labels. An *unranked ordered tree* is a tree in which nodes may have a variable number of children, with an order among them. An *XML-tree* is a relational structure $\mathcal{T}$ of signature

$$\sigma_{nav} = ((\mathsf{Lab}_L)_{L\in\Sigma}, R_{\mathsf{child}}, R_{\mathsf{next\text{-}sibling}}),$$

representing an unranked, ordered tree whose nodes are labeled using the symbols from Σ: each Lab_L, for $L \in \Sigma$, is a unary relation representing the set of nodes labeled L, R_{child} is the binary parent-child relation among nodes, and $R_{\mathsf{next\text{-}sibling}}$ is the binary immediate right-sibling relation. That is, $R_{\mathsf{child}}(x, y)$ means that y is a child of x and $R_{\mathsf{next\text{-}sibling}}(x, y)$ means that y is the immediate right-sibling of x. We say that an XML-tree $\mathcal{T}$ of signature σ_{nav} represents the navigational structure of an XML document.

An *XML document* is a structure of signature $\sigma_{dom} = \sigma_{nav} \cup \{@A_1, \ldots, @A_n\}$ over a two-sorted domain of nodes and values, where the relations from σ_{nav} over nodes are as above and the $@A_1, \ldots, @A_n$ are a fixed finite set of associated *attribute functions*, which map nodes to values. For simplicity we assume the attribute functions to be total and to take values in the integers. Partial functions can be modeled in this way, by (for example) adding a special "null" value. We use $\mathit{Node}(\mathcal{D})$ to mean the nodes of XML document $\mathcal{D}$; since $\mathcal{D}$ is usually clear from the context, we will generally write simply *Node*. Similarly, we write $\mathit{NodeSet}(\mathcal{D})$ for the set of all sets of nodes of document $\mathcal{D}$, omitting the argument $\mathcal{D}$ when it is clear.

Navigational Primitives. In XPath, the primitives employed for navigation along the tree structure of a document are called *axes*. We will consider the axes self, child, parent, descendant, descendant-or-self, ancestor, ancestor-or-self, next-sibling, following-sibling, previous-sibling, preceding-sibling, following, and finally preceding. The meaning of axis α is best given by a binary *axis relations* R_α, where R_{child} and $R_{\mathsf{next\text{-}sibling}}$ were introduced above, $R_{\mathsf{self}} = \{(n, n) : n \in \mathit{Node}\}$, $R_{\mathsf{descendant}}$ is the transitive closure of R_{child}, $R_{\mathsf{descendant\text{-}or\text{-}self}}$ is the reflexive and transitive closure of R_{child}, $R_{\mathsf{following\text{-}sibling}}$ is the transitive closure of $R_{\mathsf{next\text{-}sibling}}$. By the *inverse* of a binary relation R, we refer to the relation $\{(n', n) : R(n, n')\}$. The relations R_{parent}, R_{ancestor}, $R_{\mathsf{ancestor\text{-}or\text{-}self}}$, $R_{\mathsf{preceding\text{-}sibling}}$, and $R_{\mathsf{previous\text{-}sibling}}$ are the inverses of the relations R_{child}, $R_{\mathsf{descendant}}$, $R_{\mathsf{descendant\text{-}or\text{-}self}}$, $R_{\mathsf{next\text{-}sibling}}$, $R_{\mathsf{following\text{-}sibling}}$, respectively. Finally, $R_{\mathsf{following}}$ is the composition $R_{\mathsf{ancestor\text{-}or\text{-}self}} \circ R_{\mathsf{following\text{-}sibling}} \circ R_{\mathsf{descendant\text{-}or\text{-}self}}$ while $R_{\mathsf{preceding}}$ is the inverse of $R_{\mathsf{following}}$. We say that an axis α is the inverse of an axis β iff R_α is the inverse of R_β.

Orders among Nodes. We consider two well-known total orders on finite ordered trees. The *pre-order* $<_{\rm pre}$ and the *post-order* $<_{\rm post}$ can be defined by

$$\begin{aligned} x \;<_{\rm pre}\; y \;&:\Leftrightarrow\; R_{\mathsf{descendant}}(x,y) \vee R_{\mathsf{following}}(x,y) \\ x \;<_{\rm post}\; y \;&:\Leftrightarrow\; R_{\mathsf{descendant}}(y,x) \vee R_{\mathsf{following}}(x,y). \end{aligned}$$

Intuitively, the pre- and postorder correspond to the order in which the opening resp. closing tag of each node of a tree is seen when reading the corresponding XML document from left to right. In XML jargon, $<_{\rm pre}$ is also known as *document order* [World Wide Web Consortium 1999a].

2.2 XPath Fragments Considered in this Survey

Many results on XPath apply to the fragment that deals only with the navigational structure of an XML document. We will look at two fragments that look only at the navigational structure.

Navigational XPath and Core XPath. We define here a clean language for navigating the tag structure which we denote NavXPath. It consists of expressions whose input is a node and whose output is either a set of nodes (an element of *NodeSet*) or a Boolean. The latter are also referred to as *qualifiers* or *filters*. We will generally use $p, p' \ldots$ to vary over general XPath expressions, of any type, while $q, q' \ldots$ will be used to denote qualifiers. Expressions are built up from the grammar

$$\begin{aligned} p \;&::=\; \mathit{step} \mid p/p \mid p \cup p \\ \mathit{step} \;&::=\; \mathit{axis} \mid \mathit{step}[q] \\ q \;&::=\; p \;\mid\; \mathsf{lab}() = L \;\mid\; q \wedge q \;\mid\; q \vee q \;\mid\; \neg q, \end{aligned}$$

where *axis* stands for the axes named above, L denotes the labels in Σ, and $\wedge, \vee, \neg$ stand for *and* (conjunction), *or* (disjunction) and *not* (negation), respectively.

An expression p in NavXPath over a σ_{nav}-structure $\mathcal{D}$ is interpreted as a function $[\![p]\!]_{NodeSet}$ from a node to a set of nodes, while a qualifier q is interpreted as a unary predicate $[\![q]\!]_{Boolean} : \mathit{Node} \to \{\mathit{true}, \mathit{false}\}$. In both cases, we refer to the input node of these functions as the *context node*. The semantic functions are defined inductively on the structure of p, q. For *NodeSet* expressions p we have

(P1) $[\![\mathit{axis}]\!]_{NodeSet}(n) := \{n' : R_{axis}(n, n')\}$.
(P2) $[\![\mathit{step}[q]]\!]_{NodeSet}(n) := \{n' : n' \in [\![\mathit{step}]\!]_{NodeSet}(n) \wedge [\![q]\!]_{Boolean}(n') = \text{true}\}$.
(P3) $[\![p_1/p_2]\!]_{NodeSet}(n) := \{v : \exists w \in [\![p_1]\!]_{NodeSet}(n) \wedge v \in [\![p_2]\!]_{NodeSet}(w)\}$.
(P4) $[\![p_1 \cup p_2]\!]_{NodeSet}(n) := [\![p_1]\!]_{NodeSet}(n) \cup [\![p_2]\!]_{NodeSet}(n)$.

For qualifiers q we have

(Q1) $[\![\mathsf{lab}() = L]\!]_{Boolean}(n) := \mathsf{Lab}_L(n)$
(Q2) $[\![p]\!]_{Boolean}(n) := [\![p]\!]_{NodeSet}(n) \neq \emptyset$
(Q3) $[\![q_1 \wedge q_2]\!]_{Boolean}(n) := [\![q_1]\!]_{Boolean}(n) \wedge [\![q_2]\!]_{Boolean}(n)$
(Q4) $[\![q_1 \vee q_2]\!]_{Boolean}(n) := [\![q_1]\!]_{Boolean}(n) \vee [\![q_2]\!]_{Boolean}(n)$
(Q5) $[\![\neg q]\!]_{Boolean}(n) := \neg [\![q]\!]_{Boolean}(n)$

In the above, we have departed from standard XPath syntax in several ways: i) we have a label test as a filter, while in XPath one has testing a label as part of a step, ii) union is allowed nested arbitrarily within expressions, while in XPath it is allowed only at top-level, and iii) the set of axes includes the next-sibling and previous-sibling axes. As we will see, this gives us a fragment with nicer theoretical properties.

CoreXPath is a faithful (i.e., strictly syntactical) fragment of XPath capturing navigational properties. It is defined by making the following changes to NavXPath:

—We eliminate the filter $\mathsf{lab}() = L$ and replace the production $\mathit{step} ::= \mathit{axis} \mid \mathit{step}[q]$ by $\mathit{step} ::= \mathit{axis}::L[q] \mid \mathit{axis}::{*}[q]$, where L is a label. $\mathit{axis}::L[q]$ has the same semantics as $\mathit{axis}[\mathsf{lab}() = L][q]$ in NavXPath, while $\mathit{axis}::{*}[q]$ is the same as $\mathit{axis}[q]$ in NavXPath.

—We disallow nested union, replacing the first production by the following two: $p' ::= p \cup p \mid p$, $p ::= step \mid p/p$. p' is now the root nonterminal of the grammar.

—We remove the axes next-sibling and previous-sibling.

—We add absolute paths, $ap ::=$ "/"p, and allow them in filters, i.e. adding a production $q ::= ap$. A filter $q = /p$ has semantics $[\![q]\!]_{Boolean}(n) := [\![p]\!]_{Boolean}(n_0)$, where n_0 is the root of the document.

CoreXPath is thus properly a syntactic subset of XPath 1.0.

First-Order XPath (FOXPath). We extend CoreXPath above to allow queries that can look at the data value structure of an input document of signature σ_{dom}. FOXPath adds path expressions of the form

$$\mathrm{id}(p/@A)$$

and qualifiers of the forms

$$i \ \mathsf{RelOp}\ i \qquad\qquad p/@A\ \mathsf{RelOp}\ i \qquad\qquad p/@A\ \mathsf{RelOp}\ p'/@B$$

to the syntax of NavXPath, where p and p' are path expressions, $@A$ and $@B$ are attributes, $\mathsf{RelOp} \in \{=, \leq, <, >, \geq, \neq\}$, and i is a nonterminal denoting the constant integers.

FOXPath operates on σ_{dom}-structures with an attribute function @ID. The $\mathrm{id}(p/@A)$ expressions model the id() function of XPath, and to be fully faithful we could assume that the attribute function @ID is injective.

The semantic functions $[\![\cdot]\!]_{NodeSet} : Node \rightarrow NodeSet$ and $[\![\cdot]\!]_{Boolean} : Node \rightarrow Boolean$ of NavXPath are extended as follows to handle the additional constructs:

(P5) $[\![\mathrm{id}(p/@A)]\!]_{NodeSet}(n) := \{n' : \exists n'' \in [\![p]\!]_{NodeSet}(n)\ @\mathsf{ID}(n') = @A(n'')\}$,
(Q6) $[\![i\ \mathsf{RelOp}\ i']\!]_{Boolean}(n) := [\![i]\!]_{Int}(n)\ \mathsf{RelOp}\ [\![i']\!]_{Int}(n)$,
(Q7) $[\![p/@A\ \mathsf{RelOp}\ i]\!]_{Boolean}(n) := \exists n' \in [\![p]\!]_{NodeSet}(n)\ @A(n')\ \mathsf{RelOp}\ [\![i]\!]_{Int}(n)$, and
(Q8) $[\![p/@A\ \mathsf{RelOp}\ p'/@B]\!]_{Boolean}(n) := \exists n' \in [\![p]\!]_{NodeSet}(n)\ \exists n'' \in [\![p']\!]_{NodeSet}(n)\ @A(n')\ \mathsf{RelOp}\ B(n'')$,

where $[\![c]\!]_{Int}(n) = c$ for constant c.

Aggregate XPath (AggXPath). Next, we add on expressions to FOXPath that manipulate integers and compute aggregates.

The syntax of AggXPath is obtained from FOXPath by extending number-typed expressions i (from exclusively integer constants in FOXPath) to

$$i \ ::= \ `c' \ \mid \ i + i \ \mid \ i * i \ \mid \ \mathrm{count}(p) \ \mid \ \mathrm{sum}(p/@A)$$

where p ranges over path expressions and $@A$ is an attribute function. We call "+" and "∗" *arithmetic operators* and "count" and "sum" *aggregate operators*.

The semantic function $[\![i]\!]_{Int} : Node \rightarrow Int$ for numerical expressions of FOXPath is extended to

(I1) $[\![c]\!]_{Int}(n) := c$
(I2) $[\![i \circ i']\!]_{Int}(n) := [\![i]\!]_{Int}(n) \circ [\![i']\!]_{Int}(n) \qquad (\circ \in \{+, *\})$
(I3) $[\![\mathrm{count}(p)]\!]_{Int}(n) := |[\![p]\!]_{NodeSet}(n)|$
(I4) $[\![\mathrm{sum}(p/@A)]\!]_{Int}(n) := \Sigma\{@A(n') | n' \in [\![p]\!]_{NodeSet}(n)\}$

Aggregate XPath with position arithmetic (OrdXPath). Finally, we add the numerical operations "position()" and "last()" to AggXPath; these are called *positional operators*.

If we look at the semantic functions $[\![\cdot]\!]_{NodeSet}$, $[\![\cdot]\!]_{Int}$, and $[\![\cdot]\!]_{Boolean}$ of AggXPath, we say that they map from a context node (e.g., the root node of the document tree) to either a node set, a Boolean, or an integer value. In OrdXPath, qualifiers and numerical expressions are defined with respect to a more extensive "context" consisting of a node and two additional integers, which can be accessed by the positional operators.

(1) $[\![\cdot]\!]_{NodeSet} : \mathit{Node} \to \mathit{NodeSet}$ is as in AggXPath except for

$$(P2') \quad [\![\mathit{step}[q]]\!]_{NodeSet}(n) := \{n_j \mid [\![\mathit{step}]\!]_{NodeSet}(n) = \{n_1, \ldots, n_k\} \wedge n_1 \prec n_2 \prec \cdots \prec n_k \wedge 1 \leq j \leq k \wedge [\![q]\!]_{Boolean}(n_j, j, k)\},$$

where $\prec$ denotes either the *document order*, i.e. the total order

$$n \prec n' \Leftrightarrow R_{\mathsf{descendant}}(n, n') \vee R_{\mathsf{following}}(n, n');$$

if *step* begins with a *forward axis* (child, descendant, following, ...) or the inverse of the document order if *step* begins with any of the other axes (parent, ancestor, preceding-sibling, ...).

(2) $[\![\cdot]\!]_{Boolean} : \mathit{Node} \times \mathit{Int} \times \mathit{Int} \to \mathit{Boolean}$ is defined analogously to $[\![\cdot]\!]_{Boolean}$ of AggXPath, however taking a context consisting of a triple (n, j, k) and passing it on to all qualifier and numerical subexpressions (for instance, $[\![q_1 \wedge q_2]\!]_{Boolean}(n, j, k) := [\![q_1]\!]_{Boolean}(n, j, k) \wedge [\![q_2]\!]_{Boolean}(n, j, k)$), and

(3) $[\![\cdot]\!]_{Int} : \mathit{Node} \times \mathit{Int} \times \mathit{Int} \to \mathit{Int}$ is defined analogously to $[\![\cdot]\!]_{Int}$ of AggXPath, however passing on the full context triple (n, j, k) to its numerical subexpressions (for instance, $[\![i + i']\!]_{Int}(n, j, k) := [\![i]\!]_{Int}(n, j, k) + [\![i']\!]_{Int}(n, j, k,))$. For the new operators of OrdXPath, we have:
(I5) $[\![\text{position}()]\!]_{Int}(n, j, k) := j$
(I6) $[\![\text{last}()]\!]_{Int}(n, j, k) := k$

By *positive* FOXPath, denoted PFOXPath, (resp., NavXPath, denoted PNavXPath), we will refer to FOXPath (resp., NavXPath) without negation and inequalities (i.e., expressions $p\,\mathsf{RelOp}\,p'$ with RelOp different from "="). We say that a FOXPath query (resp., NavXPath query) is *conjunctive* (and connected) if it does not use disjunction, union, negation, or inequalities.

REMARK 2.1. The XPath fragments just presented – just like XPath 1.0 – allow for multiple qualifier brackets as part of a step expression. In all our XPath languages except for OrdXPath, this ability is redundant, since steps containing multiple qualifier brackets $axis[\cdot]\ldots[\cdot]$ can be simplified to $axis[\cdot \wedge \cdots \wedge \cdot]$. In the proofs of our survey, we will sometimes assume the simplified syntax without multiple qualifiers for convenience.

In OrdXPath this simplification is not applicable in general, and hence for this fragment the ability to use multiple qualifiers does add expressiveness.

EXAMPLE 2.2. On a context node n with three children n_1, n_2, n_3, of which the first is labeled B and the second and third are labeled A,

$$[\![\mathsf{child}[\mathsf{lab}() = A][\text{position}() = 1]]\!]_{NodeSet}(n) = \{n_2\},$$

since n_2 is the first child of n in document order that is labeled A. One can show that this query cannot be phrased with a single qualifier bracket in each step. For instance,

$$[\![\mathsf{child}[\mathsf{lab}() = A \wedge \text{position}() = 1]]\!]_{NodeSet}(n) = \{n_j \mid 1 \leq j \leq 3 \wedge [\![\mathsf{lab}() = A \wedge \text{position}() = 1]\!]_{Boolean}(n_j, j, 3)\} = \emptyset,$$

while

$$[\![\mathsf{child}[\mathsf{lab}() = A]/\mathsf{self}[\text{position}()=1]]\!]_{NodeSet}(n) =$$
$$\bigcup\{[\![\mathsf{self}[\text{position}()=1]]\!]_{NodeSet}(n_i) \mid n_i \in [\![\mathsf{child}[\mathsf{lab}() = A]]\!]_{NodeSet}(n)\} =$$
$$[\![\mathsf{self}[\text{position}()=1]]\!]_{NodeSet}(n_2) \cup [\![\mathsf{self}[\text{position}()=1]]\!]_{NodeSet}(n_3) = \{n_2, n_3\}.$$

□

The example above also shows that filters do not commute in OrdXPath.

2.3 Query Equivalence

By a *query*, we mean any expression from one of the XPath fragments introduced above. Two queries p and p' with domain *Node* are *fully equivalent* (or simply *equivalent* when it is clear from the context), denoted by $p \equiv p'$, iff for any XML document $\mathcal{D}$ and all nodes $n \in D$, $[\![p]\!]_{NodeSet}(n) = [\![p']\!]_{NodeSet}(n)$, and similarly for OrdXPath queries with context $Node \times Int \times Int$.

Let true be a shortcut for the qualifier $(\mathsf{lab}() = A) \vee \neg(\mathsf{lab}() = A)$. We say two queries are *equivalent over* Σ_0 (denoted by $\equiv_{\Sigma_0}$) where Σ_0 is a fixed finite label alphabet, if the above holds for any document $\mathcal{D}$ whose labels are in Σ_0. For example, true is equivalent to $\mathsf{lab}() = A \vee \mathsf{lab}() = B$ over the alphabet $\{A, B\}$, but not in general. We will usually work with the stronger notion of general equivalence $\equiv$, and specify when results also hold for restricted equivalence – equivalence w.r.t. some finite alphabet Σ_0.

For queries with domain *Node* (which include all NavXPath expressions), a weaker equivalence relation is defined as follows: p and p' are called *root equivalent*, denoted by $p \equiv_r p'$, iff for any XML document $\mathcal{D}$, $[\![p]\!]_{NodeSet}(rt) = [\![p']\!]_{NodeSet}(rt)$, where rt is the root of $\mathcal{D}$. For NavXPath queries defined using upward axes, root equivalence can be weaker than general equivalence: for example $\mathsf{self}[\mathsf{parent}] \equiv_r \mathsf{self}[\neg\mathsf{true}]$, since the root node has no parent, but clearly these two expressions are not fully equivalent.

2.4 Historical and Bibliographic Remarks

XPath was initially developed by James Clark and formalized and promulgated as an independent standard by the W3C starting in 1999, as XPath 1.0 [World Wide Web Consortium 1999a]. The standard defines the syntax of the language, along with use cases, but gives the semantics only informally. An early attempt to give a formal semantics is found in [Wadler 2000; 1999]. A complete and yet very concise formal semantics of XPath 1.0 can be found in [Gottlob et al. 2002].

In the process of the development of XQuery, a significant extension of XPath 1.0 was developed, released as XPath 2.0 [World Wide Web Consortium 2007]. XPath 2.0 is the result of the integration of XPath and XQuery into a common syntax and semantics definition, and its semantics is presented as part of the XQuery 1.0 Formal Semantics [World Wide Web Consortium 2002]. XPath 2.0 is a radically different language from XPath 1.0, including variables and explicit quantification. From a theoretical perspective, no polynomial time bounds can be given on basic problems like XPath 2.0 evaluation (while this is possible for XPath 1.0, see Section 4). From a practical point of view the breadth of XPath 2.0 and XQuery would require discussion to subsume nearly every aspect of general-purpose program optimization and analysis.

The extensions of XPath 2.0 over XPath 1.0 are mostly by programming language constructs that do not preserve the theoretical properties of XPath pointed out in the introduction. The largest language studied in this article, OrdXPath is a subset of XPath 1.0 (and hence, of XPath 2.0) which subsumes most of the XPath fragments for which fundamental results have been presented in the literature.

3. EXPRESSIVENESS

We now investigate where XPath "fits" in terms of other formalisms for querying trees and tree-structured data. One natural benchmark is first-order logic (FO), but we will also consider Monadic Second Order logic (MSO), the existential fragment of FO ($\exists FO$), the positive existential fragment of FO ($\exists^+ FO$) and the fragment

FO^k of FO formulas that use at most k distinct variables. The semantics of these languages is standard [Libkin 2004]. For a logical language $\mathcal{L}$, we will use $\mathcal{L}[\sigma]$ to denote the formulas of $\mathcal{L}$ over vocabulary σ. We discuss our choice of predicate logics as a benchmark, and mention alternatives, at the end of this section.

3.1 Expressiveness of NavXPath and CoreXPath

We start by investigating how NavXPath and CoreXPath compare to first-order logic over the navigational structure of XML documents, and to each other. Note that a formula of first-order logic with two free variables can be thought of as defining a mapping from *Node* to *NodeSet*, while a formula with one free variable defines a mapping from *Node* to Boolean. We say that a Boolean query q in one of our XPath fragments is fully equivalent to a first-order formula $\phi(x)$ if for any XML document $\mathcal{D}$ and all nodes $n \in D$, $[\![p]\!]_{Boolean}(n) \leftrightarrow \mathcal{D} \models \phi(n)$. We say that a nodeset query p in one of our XPath fragments is fully equivalent to a first-order formula $\phi(x, y)$ if for any XML document $\mathcal{D}$ and all nodes $m, n \in D$, $n \in [\![p]\!]_{NodeSet}(m) \leftrightarrow \mathcal{D} \models \phi(m, n)$.

The semantics of NavXPath presented in Section 2.2 already gives a translation into these first-order languages.

Recall that $\sigma_{transnav}$ is the vocabulary extending σ_{nav} with $R_{\mathsf{descendant}}$ and $R_{\mathsf{following\text{-}sibling}}$. Then,

Proposition 3.1. *For every* NavXPath *expression e one can find (in linear time) a corresponding formula ϕ in $FO[\sigma_{transnav}]$ fully equivalent to e. Furthermore,*

—$\phi \in FO[(\mathsf{Lab}_L)_{L\in\Sigma}, R_{\mathsf{child}}]$ *if e uses only* child *and* parent *axes,*

—$\phi \in FO[(\mathsf{Lab}_L)_{L\in\Sigma}, R_{\mathsf{descendant}}]$ *if e uses only upward and downward axes, and*

—$\phi \in FO[\sigma_{nav}]$ *if e uses only* child, parent, next-sibling, previous-sibling.

CoreXPath can be translated into NavXPath in linear time, just by expanding out the definitions. Hence this proposition holds for CoreXPath as well. Note also that this proposition holds both for path expressions *returning* nodesets (in this case ϕ has two free variables) and for those returning Boolean expressions (here ϕ has one free variable).

However, this is not an *exact* characterization of the expressiveness of NavXPath. It is easy to find first-order queries over trees that are not expressible in NavXPath: for example, the query that asks whether the tree has two nodes labeled C that are in an ancestor relationship, and such that all nodes between them are labeled B. We now show that NavXPath does have an exact characterization, corresponding precisely to two-variable logic.

We first work on characterizing NavXPath nodeset queries. To do this we introduce a normal form for queries with two free variables that are built from FO^2 formulas in one free variable. over vocabulary $\sigma_{transnav}$. *XPNF* is the set of queries that are disjunctions of $\sigma_{transnav}$ formulas $\gamma(z_1, z_n)$ of the form:

$$\exists z_2 \ldots \exists z_{n-1}\ \rho_1(z_1) \wedge \chi_1(z_1, z_2) \wedge \rho_2(z_2) \wedge \ldots \wedge \chi_{n-1}(z_{n-1}, z_n) \wedge \rho_n(z_n)$$

where the z_i here are distinct variables, the ρ_i are FO^2 formulae, and the $\chi_i(z_i, z_{i+1})$ are disjunctions of binary atomic formulas over predicates from $\sigma_{transnav}$.

Theorem 3.2 [Marx and de Rijke 2004]. NavXPath *corresponds to FO^2 in expressiveness, in the following sense.*

—*For every* NavXPath *expression returning a Boolean there is a corresponding fully equivalent expression in FO^2 over the signature $\sigma_{transnav}$, and for every FO^2 expression there is a corresponding fully equivalent* NavXPath *expression.*

—*For every* NavXPath *expression returning a NodeSet, there is a corresponding expression in XPNF and vice versa.*

Proof (Sketch). We first show the direction from NavXPath *NodeSet* expressions to *XPNF* and from NavXPath Boolean expressions to FO^2. We will restrict to *unnested* NavXPath *expressions*, that is, NavXPath expressions that have union only at top-level. These have the same expressiveness as general NavXPath expressions. Since the target classes FO^2 and *XPNF* are closed under disjunction, it suffices to translate expressions that have no occurrence of the union operator. So it suffices to show that all NavXPath *NodeSet* expressions that do not use the union operator translate to *XPNF* expressions without top-level disjunction, and every NavXPath Boolean expression that does not use the union operator translates to an FO^2 expression. We show this pair of statements by simultaneous induction. The base case for $\mathsf{lab}() = A$ is simple, as is the case for Boolean operations in Boolean expressions (since FO^2 is closed under Boolean operators). The case *step*$[q]$ can be translated into *XPNF* formula $\chi(x, y) \wedge \phi(y)$, where χ is a *XPNF* formula without top-level disjunction formed inductively for *step*, and ϕ is an FO^2 formula formed for q. We now do the inductive proof for $p = p_1/p_2$. By induction, we assume we have *XPNF* formulas (without top-level disjunction) γ_1 equivalent to p_1 and γ_2 equivalent to p_2. If we have

$$\gamma_1 = \exists z_2 \ldots \exists z_{m-1} \ \Big(\bigwedge_{i=1}^{m-1} \rho'_i(z_i) \wedge \chi_i(z_i, z_{i+1})\Big) \wedge \rho'_m(z_m)$$

and

$$\gamma_2 = \exists z_m \ldots \exists z_{n-1} \ \Big(\bigwedge_{i=m}^{n-1} \rho''_i(z_i) \wedge \chi_i(z_i, z_{i+1})\Big) \wedge \rho''_n(z_n)$$

then we can write γ_1/γ_2 as

$$\exists z_2 \ldots \exists z_{n-1} \Big(\bigwedge_{i=1}^{n-1} \rho_i(z_i) \wedge \chi_i(z_i, z_{i+1})\Big) \wedge \rho_n(z_n) \qquad (1)$$

where $\rho_i(z_i)$ is $\rho'_i(z_i)$ for $i < m$, $\rho'_i(z_i) \wedge \rho''_i(z_i)$ for $i = m$, and $\rho''_i(z_i)$ for $i > m$.

The other interesting inductive case is that of qualifiers of the form p. By induction we have a *XPNF* formula γ representing p. We will assume $\gamma(z_1, z_n)$ to be as shown in equation (1).

We need to show that the formula $\exists z_n \gamma(z_1, z_n)$ is in FO^2. Suppose that n is odd (the case where n is even is similar). Let $var(i) = z_1$ for i odd and z_2 for i even. Let $\phi([x \mapsto y])$ denote the formula obtained by substituting all occurrences of variable x by y in ϕ. Define $\psi_n = \rho_n([z_n \mapsto var(n)])$ and $\psi_{i-1} = \rho_{i-1}([z_{i-1} \mapsto var(i-1)]) \wedge \exists var(i)\ \chi_i(var(i-1), var(i)) \wedge \psi_i$. Then ψ_i is an FO^2 sentence with $var(i)$ free. We can verify that ψ_1 is equivalent to $\exists z_n \gamma(z_1, z_n)$.

The converse direction is to show by induction that formulas in *XPNF* can be translated to NavXPath *NodeSet* expressions, while FO^2 formulas with one free variable can be translated to NavXPath Boolean expressions. Since the first statement follows easily from the second, we focus on the proof of the second. The translation function T is formed by induction on the structure of an FO^2 formula. The atomic cases are straightforward, as are the Boolean operations. The interesting case is $\exists y\, \beta(x, y)$, where β is in FO^2. Formula β can be assumed to be a Boolean combination of atomic binary formulas and FO^2 formulas in one free variable of lower quantifier rank. Let β' be a formula equivalent to β obtained by turning β into a Disjunctive Normal Form (DNF) over formulas of the two forms above, and then replacing each disjunct $\phi(x, y)$ that does not contain a binary atom by $(\phi(x, y) \wedge x = y) \vee (\phi(x, y) \wedge x \neq y)$. This replacement preserves the DNF.

The atomic binary predicates in β' are either equality, inequality, or axis relations; however, equality $x = y$ can be replaced by $\mathsf{self}(x, y)$, and an inequality $x \neq y$ can be replaced by a disjunction of four axis relations (y is either and ancestor or descendant of x or follows or precedes x). Let β'' be obtained by applying these substitutions to β' and again turning the formula into DNF.

Since two axis predicates are either inconsistent with one another (i.e., the axis relations have an empty intersection) or subsume each other, we can assume $\beta''(x, y)$ to be of the form

$$\bigvee_i \phi_i(x) \wedge R_{\chi_i}(x, y) \wedge \psi_i(y),$$

that is, each disjunct contains precisely one binary atom.

We can easily translate $\beta''(x,y)$ into NavXPath as

$$T(\beta'') ::= \bigcup_i \mathsf{self}[T(\phi_i)]/\chi_i[T(\psi_i)].$$

□

We note that the argument from NavXPath to FO^2 shows that there is a polynomial time translation from unnested NavXPath to FO^2; for general NavXPath expressions the best translation we know of is in exponential time. This mapping introduces atomic predicates in the output corresponding only to axes mentioned in the input; hence NavXPath filters without the next-sibling or previous-sibling axes map to FO^2 formulas that do not use (atomic relations for) these axes.

In the direction from FO^2 to NavXPath, the translation also yields an output that is exponential in the input in the worst case, and this has been shown to be unavoidable. See [Marx and de Rijke 2004] for discussion and proof of this; we will give a further argument that there is no polynomial translation in Section 5.[1] This direction does introduce new axes. The sibling axes may appear in the output even when the original formula mentions only the child axis; the *XPNF* formula $x \neq y$ cannot be translated into NavXPath unless the sibling axis is present. Similarly, transitive axes are introduced in the translation.

On the other hand, next-sibling and previous-sibling are not introduced in this translation unless the corresponding atomic predicates occur in the input. Since next-sibling and previous-sibling are not introduced in either direction, we have that NavXPath filters without these axes correspond exactly to FO^2 formulas that do not have atomic relations for these axes. Since CoreXPath expressions are, up to syntactic sugar, exactly those NavXPath expressions that do not include the non-transitive sibling axes, we have:

THEOREM 3.3. CoreXPath *corresponds in expressiveness to two-variable logic over the vocubulary formed by removing the relation* $R_{\mathsf{next\text{-}sibling}}$ *from* $\sigma_{transnav}$.

From these two results and prior known results about FO^2, we obtain:

PROPOSITION 3.4. *There are queries expressible in* NavXPath *(and hence in* FO^2*) that are not expressible in* CoreXPath.

PROOF. If we restrict to trees of depth 2, all axes collapse to sibling axes, and hence CoreXPath corresponds to FO^2 with only the transitive sibling axes while NavXPath corresponds to all sibling axes. Taking the natural correspondence between trees of depth 2 correspond and words, CoreXPath maps to FO^2 with only the linear order relation, while NavXPath corresponds to FO^2 with successor and linear order. But it is known that a successor relation of a linear order cannot be expressed in FO^2 over the signature whose only binary predicate is for the linear order (see e.g. Section 7 of [Thérien and Wilke 1998]). □

We now turn to the consequences of this characterization for closure properties of NavXPath and CoreXPath. It is clear that NavXPath qualifiers are closed under

[1]Although the argument there is relative to a complexity-theoretic assumption.

Boolean operations, since we have explicit operators for these; it can also be seen to follow from Theorem 3.2, since FO^2 is obviously Boolean closed. What about the closure properties of NavXPath expressions? In [Marx 2005], the following is shown:

THEOREM 3.5 [MARX 2005]. NavXPath *and* CoreXPath *expressions returning nodesets are closed under intersection and union, but not under complement.*

Closure under union is obvious, since NavXPath has a built-in union operator. Closure under intersection follows from the fact that the conjunction of *XPNF* queries can be rewritten as a conjunction of atomic $\sigma_{transnav}$ formulas and a single FO^2 formula. Every conjunctive query on trees can be transformed into an equivalent union of acyclic conjunctive queries [Benedikt et al. 2003; Gottlob et al. 2004] (cf. Theorem 3.9 below), and unions of acyclic conjunctive queries can be easily translated into NavXPath. The same argument holds for CoreXPath.

The lack of closure under complementation may seem surprising. In fact, [Marx 2005] shows a stronger result: any extension of NavXPath closed under complementation can express all first-order properties. The proof is by showing that an "until" operator can be defined by complementing NavXPath expressions. The following example is taken from page 7 of [Marx 2005]: Let $\phi(x, y)$ hold iff y is an A-labeled descendant of x and every descendant of x that is an ancestor of y is labeled B. Then ϕ is expressible in NavXPath extended with a complement operator $(\cdot)^c$ as:

$$\mathsf{descendant[lab()} = A)] \cap (\mathsf{descendant[lab()} \neq B]/\mathsf{descendant})^c$$

Above, we use also the intersection operator $\cap$, but this can easily be defined using complementation and union.

The translation of unnested NavXPath to FO^2 can be extended as follows: let NavXPath$^\cap$ be the extension of NavXPath with the intersection operator $\cap$, and let *unnested* NavXPath$^\cap$ be the same but with union allowed only at top-level. By Theorem 3.5 above, we have NavXPath$^\cap$ has the same expressiveness as NavXPath (for both expressions and qualifiers). Hence NavXPath$^\cap$ qualifiers have the same expressiveness as FO^2 formulas. Using the argument of [Olteanu et al. 2002], one can show that even unnested NavXPath$^\cap$ formulas can be exponentially more succinct than NavXPath formulas. However, unnested NavXPath$^\cap$ formulas can still be translated into FO^2 efficiently:

PROPOSITION 3.6. *There is a polynomial time function taking an unnested* NavXPath$^\cap$ *filter and producing a FO^2 formula $\phi(x)$ fully equivalent to it.*

Proof. We extend the dual translations from the proof of Theorem 3.2 to go from NavXPath$^\cap$ *NodeSet* expressions without union to *XPNF* queries and from NavXPath Boolean expressions without union to FO^2 queries. We use exactly the same construction of a translation function, let us call it f, as for NavXPath, but for the inductive step for $f(E_1 \cap E_2)$ we translate into $f(E_1) \wedge f(E_2)$. □

We now provide an example of a navigational FO query that we prove not to be expressible in NavXPath. Our example, a new immediately-following axis, has a practical motivation. Computational linguists have proposed the addition of such an axis to XPath to ask practical queries on linguistic trees [Bird et al. 2005]. We can give a semantics to this axis using a corresponding binary relation $R_{\mathsf{immediately\text{-}following}}$, which holds of (x, y) iff

$$R_{\mathsf{following}}(x, y) \wedge \neg\exists z\ (R_{\mathsf{following}}(x, z) \wedge R_{\mathsf{following}}(z, y)).$$

In [Bird et al. 2005] an extension of XPath with immediately-following is proposed. We show here the following:

PROPOSITION 3.7. *There is no* NavXPath *expression E fully equivalent as a nodeset query to* immediately-following.

Proof. Consider documents that include a chain of A elements starting from the root to a leaf, with one of the following holding for each element x in the chain:

(1) x has a single A child (the next element of the chain), and no other children,
(2) x has no children (i.e. it is the lowest element of the chain),
(3) x has a single A child and a single B child, or
(4) x has a single A child and a single C child.

It is easy to construct a NavXPath qualifier Q_0 that holds of the root of a document iff the document is of the above form. Consider the qualifier Q_1

$$\mathsf{lab}() = A \wedge \mathsf{immediately\text{-}following}[\mathsf{lab}() = B]$$

in NavXPath extended with immediately-following.

That is, Q_1 holds of an A node iff it has an immediately-following node that is a B. For a node n in a tree whose root satisfies Q_0, Q holds at n iff the first ancestor of n which has a non-A child has a B child. We claim that there is no NavXPath qualifier equivalent to $Q_1 \wedge Q_0$. From this, the proposition follows. From Theorem 3.2, it suffices to show that no two-variable logic formula can express $Q_1 \wedge Q_0$.

We will reduce expressibility of $Q_1 \wedge Q_0$ over trees to a statement about expressibility of a certain property in two-variable logic over strings. Let FO^- be the logic built up using quantification only over A nodes, where the vocabulary includes the binary predicates $R_{\mathsf{descendant}}$ and R_{child} and unary predicates P_1, P_2, P_3, P_4, where P_i holds of x iff case i holds above.

CLAIM 3.8. *For every $FO[\sigma_{transnav}]$ sentence $\phi(x)$ there is an FO^- sentence $\phi^-(x)$ with the same number of variables as ϕ which is equivalent to ϕ over all A-nodes within all documents whose root satisfies Q_0.*

Informally, ϕ^- is obtained inductively by replacing variables over B, C nodes by variables over their A parents. A sentence $\phi = \exists x\ B(x)$ would map to $\phi^- = \exists x \in A\ P_3(x)$. Formally, we proceed as follows. Let $\mathsf{SecChild}(D, x)$ be the partial function on nodes of D that maps a node labeled A to its second child, if such a child exists, and $\mathsf{Self}(D, x)$ be the identity function on nodes labeled A. We create a function $T(\phi, b)$ for $\phi \in FO[\sigma_{transnav}]$, b a function from the free variables of ϕ to either SecChild or Self, returning a formula $\phi' \in FO^-$ with the same free variables as ϕ, and such that: for all documents D, $T(\phi(x, y), b)$ holds of A nodes m, n iff $\phi(x, y)$ holds when applied to $b(D, m), b(D, n)$, and similarly for $\phi(x), \phi(y)$.

The main atomic cases for T are:

—$T(R_{\mathsf{next\text{-}sibling}}(x, y), b)$ is $(P_3(y) \vee P_4(y)) \wedge R_{\mathsf{child}}(y, x)$ if $b(x) = \mathsf{Self}$ and $b(y) = \mathsf{SecChild}$, and is $false$ otherwise.
—$T(R_{\mathsf{child}}(x, y), b)$ is $R_{\mathsf{child}}(x, y)$ if $b(x) = \mathsf{Self}$ and $b(y) = \mathsf{Self}$, is $(P_3(x) \vee P_4(x)) \wedge x = y$ if $b(x) = \mathsf{Self}$ and $b(y) = \mathsf{SecChild}$, and is $false$ otherwise.
—$T(R_{\mathsf{descendant}}(x, y), b)$ is $R_{\mathsf{descendant}}(x, y)$ if $b(x) = \mathsf{Self}$ and $b(y) = \mathsf{Self}$, is $P_3(y) \vee P_4(y)$ if $b(x) = \mathsf{Self}$ and $b(y) = \mathsf{SecChild}$, and is $false$ otherwise.
—$T(B(x), b)$ is $P_3(x)$ if $b(x) = \mathsf{SecChild}$, and is $false$ otherwise.
—$T(C(x), b)$ is $P_4(x)$ if $b(x) = \mathsf{SecChild}$ and is $false$ otherwise.
—$T(A(x), b)$ is $true$ if $b(x) = \mathsf{Self}$, and is $false$ otherwise.

The other atomic cases are similar. The inductive cases are:

—$T(\exists x \rho(x, y), b) = \bigvee_{b' : b'|\{y\} = b} \exists x \in A\ T(\rho(x, y), b')$
—$T(\forall x \rho(x, y), b) = \bigwedge_{b' : b'|\{y\} = b} \exists x \in A\ T(\rho(x, y), b')$
—$T(\phi_1 \wedge \phi_2, b) = T(\phi_1, b) \wedge T(\phi_2, b)$
—$T(\phi_1 \vee \phi_2, b) = T(\phi_1, b) \vee T(\phi_2, b)$

—$T(\neg\phi, b) = \neg T(\phi, b)$

Finally, for a sentence we let $\phi^-(x)$ be $\bigvee_b T(\phi(x), b)$, where in the disjunction b ranges over all the bindings for x. One can verify inductively that T, and hence ϕ^- has the required properties.

From this construction, we see that if $\phi(x) \in \mathsf{NavXPath}$ expresses $Q_0 \wedge Q_1$, then $\phi^-(x)$ must hold of an A-node n iff the first ancestor of n which satisfies $P_3 \vee P_4$ satisfies P_3. Let S_0 be the set of strings from alphabet $\Sigma = \{P_1, P_2, P_3, P_4\}$, ending with the symbol P_1. There is an obvious bijection F from documents whose root satisfies Q_0 to strings in S_0. Using this function, we can see that $\phi^-(x)$, considered as a predicate on strings in S_0, holds at node n iff the first ancestor of n which satisfies $P_3 \vee P_4$ satisfies P_3. But then by flipping the variables in every predicate $R_{\mathsf{descendant}}$ or R_{child} in ϕ^- we obtain a two-variable formula $\phi^-(x)$ that holds at node n of string s iff the first descendant of n satisfying $P_3 \vee P_4$ satisfies P_3. From this we easily get a contradiction of prior results about the inexpressibility of the Until operator in two variable logic (for strings, those of [Etessami and Wilke 2000; Etessami et al. 2002], or for trees those of [Marx 2004b]). Consider the query Q that holds of a string s iff s has a substring that contains two nodes satisfying P_3 but none satisfying P_4. If $\phi^-(x)$ were expressible in two-variable logic, then Q would be expressible over strings in two-variable logic over the vocabulary consisting of the labels, the descendant predicates, and the child predicate. But in [Etessami and Wilke 2000] it is shown that Q (denoted there by $FAIR_2$) is not expressible in Unary Temporal Logic, and by [Etessami et al. 2002] Unary Temporal Logic is the same as two-variable logic over strings. Hence Q is not expressible in two-variable logic, and we have a contradiction. □

Note that the problem of deciding whether a given FO sentence over trees is in NavXPath (i.e. is a two-variable sentence in $\sigma_{transnav}$) is still open, as is the membership problem for CoreXPath. The analogous problem for strings (membership in FO^2) is known to be decidable [Beauquier and Pin 1989].

3.2 Expressiveness of Fragments of NavXPath

NavXPath is still a large language, and many applications make use only of the positive fragment.

Following [Benedikt et al. 2003], we characterize NavXPath both using logic and a visual query formalism, *tree patterns.*

A tree pattern (over label alphabet Σ) is a node and edge-labeled tree. Edges are labeled with a forward axis (child, descendant, following-sibling). In a *Boolean tree pattern* node labels have one component that is either a label from Σ or wildcard and another component that identifies whether a node is the distinguished *context node* or not. In a *unary tree pattern* the additional component identifies a node as either the context node, the selected node, or neither. Figure 1 shows a unary tree pattern. Following the standard convention for drawing patterns, double lines are used for a descendant edge and single lines for a child edge. A star is used to denote the selected node, and the context node is implicitly the root node. A Boolean pattern corresponds to a Boolean query, returning true at context node n in a document iff there is a homomorphism from the pattern to the document mapping the context to n. A unary tree pattern corresponds to a *NodeSet* query, which returns node n' on input n iff there is a homomorphism from the pattern to the document which maps a node labeled context to n and the selected node to n'. The pattern in the figure is equivalent to the XPath expression

$$\mathsf{self}::A[\mathsf{child}::B][\mathsf{descendant}::D]/\mathsf{child}::C$$

A finite set of tree patterns can be considered as a query, returning the union of the results of the individual patterns in the case of unary tree patterns, and

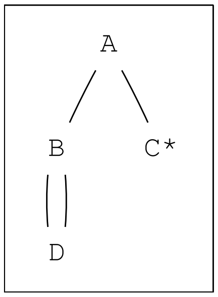


Fig. 1. Tree pattern

returning the disjunction of the results in the case of Boolean tree patterns.

THEOREM 3.9. *The following have equal expressiveness (up to full equivalence)*

—PNavXPath *NodeSet queries,*

—$\exists^+ FO$ *formulas* $\phi(x, y)$ *in the signature* $\sigma_{transnav}$*, and*

—*sets of unary tree patterns.*

A similar result holds for negation-free CoreXPath, but where the formulas do not include $R_{\textsf{next-sibling}}$. Note that this result is incomparable to Theorem 3.2. Theorem 3.2 applies to arbitrary NavXPath, and says that they are fully equivalent to acyclic conjunctive queries over atoms that include arbitrary FO^2 formulas, possibly with negation. This result applies only to PNavXPath queries, but states that that they can be written as conjunctions of only atomic formulas, where the the conjunction must constrain the variables to be "tree-like".

We give a sketch of why the above holds: further details (for the case where there are only upward or downward axes, but no sideways axes such as following or following-sibling) can be found in [Benedikt et al. 2003]; the general case is proved in [Gottlob et al. 2004]. For every PNavXPath *NodeSet* query, and unary tree patterns, the corresponding equivalent $\exists^+ FO$ formula can be found in linear time, simply by translating the semantics of PNavXPath or of tree patterns into logic. Translating from unary tree pattern queries to PNavXPath queries is likewise straightforward: path steps are used to traverse the path from the context node upward to the least common ancestor of the context and selected node, then downwards from this ancestor to the selected node. The existence of subtrees sprouting off from this path is asserted using filters. Translation of $\exists^+ FO$ formulas into tree patterns is done by first translating them into acyclic positive queries, which immediately correspond to forests of tree patterns:

LEMMA 3.10 [OLTEANU ET AL. 2002; BENEDIKT ET AL. 2003; GOTTLOB ET AL. 2004]. *For every conjunctive query over trees there is an equivalent acyclic positive query. This query can be computed in exponential time.*

Proof. For notational simplicity, we will assume that the input query $\exists x_1 \cdots x_k\, Q$ ($k \geq 0$), with Q a conjunction of atomic formulas that uses variables $x_1, \ldots, x_k$, is Boolean. The proof, however, immediately generalizes to conjunctive queries of arbitrary arity. W.l.o.g., we assume that Q contains no $R_{\textsf{following}}$-atoms. (Each atom $R_{\textsf{following}}(x, w)$ can be rewritten using $R^*_{\textsf{child}}$ and $R^+_{\textsf{next-sibling}}$ atoms as $R^*_{\textsf{child}}(x, y) \wedge R^+_{\textsf{next-sibling}}(y, z) \wedge R^*_{\textsf{child}}(z, w)$, where y and z are new variables.)

Consider the conjunctive normal form formula

$$\phi := \bigwedge_{1 \leq i < j \leq k} (x_i = x_j \vee x_i \ <_{\text{pre}} \ x_j \vee x_j \ <_{\text{pre}} \ x_i).$$

$R \setminus S$	R_{child}	R^+_{child}	$R_{\mathsf{next\text{-}sibling}}$	$R^+_{\mathsf{next\text{-}sibling}}$
R_{child}	unsat	unsat	sat	sat
R^+_{child}	sat	sat	sat	sat
$R_{\mathsf{next\text{-}sibling}}$	unsat	unsat	unsat	unsat
$R^+_{\mathsf{next\text{-}sibling}}$	unsat	unsat	sat	sat

Table I. Satisfiability of $R(x,z) \wedge S(y,z) \wedge x <_{\text{pre}} y$ for pairs of axes R, S.

Let Ψ be the set consisting of the $3^{\binom{k}{2}}$ disjuncts of the disjunctive normal form of ϕ. For $\psi \in \Psi$ let Q_ψ be the conjunction of atomic formulas obtained from $Q \wedge \psi$ by the following steps, in the indicated order.

(1) We remove all occurrences of equality atoms $x = y$ in arbitrary order and replace, for each such atom, all occurrences of y by x.
(2) For $R \in \{R_{\mathsf{child}}, R_{\mathsf{next\text{-}sibling}}\}$, we remove all atoms $R^*(x,x)$ from Q_ψ and replace all occurrences of $R^*(x,y)$ (where x and y are different variables) by $R^+(x,y)$. The latter is an equivalent rewriting since Q_ψ contains either atom $x <_{\text{pre}} y$ or $y <_{\text{pre}} x$, thus x and y must map to different nodes.
(3) For $R \in \{R_{\mathsf{child}}, R_{\mathsf{next\text{-}sibling}}\}$, if Q_ψ contains atoms $R(x,y)$, $R^+(x,y)$ then $R^+(x,y)$ is removed from Q_ψ.

Observe that the binary atoms or Q_ψ use only R_{child}, R^+_{child}, $R_{\mathsf{next\text{-}sibling}}$, $R^+_{\mathsf{next\text{-}sibling}}$, and $<_{\text{pre}}$ as predicates. We can verify that $\exists \vec{x}\, Q_\psi$ is true if and only if $\exists \vec{x}\, Q \wedge \psi$.

Let $\mathbf{Q} = \{\exists \vec{x}\, Q_\psi \mid \psi \in \Psi\}$. Then

$$Q \;\equiv\; \exists \vec{x}\, Q \wedge \phi \;\equiv\; \bigvee \{\exists \vec{x}\, Q \wedge \psi \mid \psi \in \Psi\} \;\equiv\; \bigvee \mathbf{Q}.$$

In the following, we will call the binary relation E with

$$xEy \;:\Leftrightarrow\; \text{there is an atomic formula } R(x,y) \text{ in } Q_\psi$$

(with R a binary predicate – either an axis or $<_{\text{pre}}$) the *graph of* Q_ψ. Note that E is either cyclic or defines a total order on the variables in Q_ψ because there is an edge between any two variables of Q_ψ.

Now, for each Q_ψ of $\mathbf{Q}$, we repeat the following steps until we terminate:

—If the graph of Q_ψ is cyclic, Q_ψ is unsatisfiable and is removed from $\mathbf{Q}$. Termination. Otherwise, the graph of Q_ψ is acyclic and thus constitutes a total order of the variables in Q_ψ.
—If Q_ψ contains atoms $R(x,y), S(x,y)$ where $R \in \{R_{\mathsf{child}}, R^+_{\mathsf{child}}\}$ and $S \in \{R_{\mathsf{next\text{-}sibling}}, R^+_{\mathsf{next\text{-}sibling}}\}$, Q_ψ is unsatisfiable and is removed from $\mathbf{Q}$. Termination.
—If there are no two atoms $R(x,z), S(y,z)$ in Q_ψ with x and y distinct variables and R, S different from $<_{\text{pre}}$ then Q_ψ is acyclic. Termination.
—We choose the pairs of atoms $R(x,z), S(y,z)$ (x and y distinct variables and R, S different from $<_{\text{pre}}$) such that z is maximal with respect to the total order given by the graph of Q_ψ. From among these, we choose a pair such that x is minimal with respect to the total order. By our choice, $x <_{\text{pre}} y$ is in Q_ψ. If $R(x,z) \wedge S(y,z) \wedge x <_{\text{pre}} y$ is unsatisfiable (the unsatisfiable cases can be found in Table I), remove Q_ψ from $\mathbf{Q}$ and terminate. Otherwise, replace atom $R(x,z)$ by $R(x,y)$.

The above algorithm terminates because there are no more than $\binom{k}{2}$ non-$<_{\text{pre}}$-atoms and whenever we replace an atom $R(x,z)$ by an atom $R(x,y)$, y is smaller than z with respect to the total order. Once we have processed a pair of atoms $R(x,z)$, $S(y,z)$, we never have to process pairs of atoms $R'(x,z)$, $S'(y',z)$ for the same x and z again. Thus processing a single Q_ψ takes polynomial time and the complete rewriting of Q takes exponential time.

It can be verified that replacing $R(x,z)$ in the satisfiable cases of $R(x,z) \wedge S(y,z) \wedge x <_{\text{pre}} y$ by $R(x,y)$ is an equivalent rewriting:

—$R = R^{+}_{\mathsf{child}}$, $S \in \{R_{\mathsf{child}}, R^{+}_{\mathsf{child}}\}$: if x and y are ancestors of z, then $x <_{\text{pre}} y$ implies that x is an ancestor of y.

—$R = R^{+}_{\mathsf{next\text{-}sibling}}$, $S \in \{R_{\mathsf{next\text{-}sibling}}, R^{+}_{\mathsf{next\text{-}sibling}}\}$: analogous.

—$R \in \{R_{\mathsf{child}}, R^{+}_{\mathsf{child}}\}$, $S \in \{R_{\mathsf{next\text{-}sibling}}, R^{+}_{\mathsf{next\text{-}sibling}}\}$: Since x is a parent/ancestor of z and y is a left sibling of z, x is also a parent/ancestor of y.

Each conjunctive query Q_ψ in the set $\mathbf{Q}$ obtained as described above is acyclic if all the $<_{\text{pre}}$-atoms are removed. Doing just that is an equivalent rewriting: Let Q'_ψ be the conjunction of atoms of Q_ψ excluding the $<_{\text{pre}}$-atoms of Q_ψ. Then $\exists \vec{x}\, Q_\psi \subseteq \exists \vec{x}\, Q'_\psi \subseteq \exists \vec{x}\, Q$; thus, $\exists \vec{x}\, Q \equiv \bigvee \mathbf{Q} \subseteq \bigvee \{\exists \vec{x}\, Q'_\psi \mid Q_\psi \in \mathbf{Q}\} \subseteq \exists \vec{x}\, Q$. □

The translations from PNavXPath into FO^2 and from tree pattern queries into both PNavXPath and (hence) FO^2 are linear, but every other translation in the above theorem is exponential in the worst case; from $\exists^+ FO$ to PNavXPath and from $\exists^+ FO$ to tree patterns, this is shown in [Gottlob et al. 2004]. For the translation from PNavXPath to tree patterns, note that PNavXPath can encode a Conjunctive Normal Form of a propositional formula (e.g. proposition p_i encoded by $[R_{\mathsf{child}}/[\mathsf{lab}() = A_i]]$). A set of tree patterns would correspond to a Disjunctive Normal Form representation of the same formula. Since it is known that there is an exponential blow-up in going from CNF to DNF, the exponential blow-up of this translation follows.

A similar argument gives:

Theorem 3.11. *The following have equal expressiveness (up to full equivalence)*

—*Boolean* PNavXPath *queries,*

—*$\exists^+ FO$ formulas $\phi(x)$ in the signature $\sigma_{transnav}$,*

—*$\exists^+ FO$ formulas $\phi(x)$ in the signature $\sigma_{transnav}$ with at most two variables, and*

—*sets of Boolean tree patterns.*

It is easy to show that $\exists^+ FO[\sigma_{transnav}]$ is closed under intersection and union, but not complement. From this and the theorem above, one has:

Corollary 3.12. *Boolean* PNavXPath *queries are closed under intersection and union, but not under complementation.*

Another consequence of the above is:

Corollary 3.13 [Olteanu et al. 2002]. *For every* PNavXPath *query p, there is a query p' that contains none of the axes* preceding-sibling, previous-sibling, *and is equivalent to p. In addition there is a query p' containing none of the "backward axes" (*parent, ancestor, ancestor-or-self, preceding-sibling, previous-sibling*) such that $p \equiv_r p'$.*

To see this, consider the translation of a tree pattern into PNavXPath. This translation can be done in such a way as to never introduce preceding-sibling or previous-sibling. The upward axes parent and ancestor are introduced only when the context node in the pattern is not the root. But under root equivalence, a tree pattern can always be taken to have the context node to the root (since otherwise the pattern is root equivalent to true).

[Olteanu et al. 2002] gives a rewrite system that removes the backward axes (parent, ancestor, ancestor-or-self, preceding-sibling), assuming root equivalence.

It is known that upward axes and backward axes cannot be removed in the presence of negation or data values: for negation, one can consider the query $p =$ descendant[lab() $= B \wedge \neg$ancestor[lab() $= A$]]. One can show by an analysis of

NavXPath queries without upward axes that this cannot be expressed without the use of ancestor.

3.3 Expressiveness of FOXPath

Much less is known about the expressiveness of FOXPath and AggXPath than for NavXPath. It is easy to see that FOXPath expressions can be translated into first-order logic over the signature

$$\sigma^{+}_{val} = \sigma_{nav} \cup \{\mathsf{RelOp}_{@A_i,@A_j} \mid i,j \in \{1,\ldots,n\}, \mathsf{RelOp} \in \{=,\neq,<,\leq,>,\geq\}\} \\ \cup \{R_{\mathsf{descendant}}, R_{\mathsf{following\text{-}sibling}}\},$$

where $\mathsf{RelOp}_{@A_i,@A_j}(x,y)$ holds of nodes x and y iff $x.A_i$ RelOp $y.A_j$. An important observation is the following, analogous to one direction of Theorem 3.2:

Proposition 3.14. *Every* FOXPath *expression p can be translated (in linear time) to a fully equivalent formula ϕ_p over vocabulary σ^{+}_{val} such that ϕ_p uses at most three variables. In case p is a Boolean expression, p will have one free variable, and in case p is a NodeSet expression it will have two free variables.*

Proof. The translation is inductive; the only new case over NavXPath is the case of a qualifier $F = E$ RelOp E'. Letting $\phi_E(x,y), \phi_{E'}(x,y)$ be the translations formed inductively from E, E' respectively. Then we can set

$$\phi_F = \exists y\ \exists y'\ \phi_E(x,y) \wedge \phi_{E'}(x,y') \wedge \mathsf{RelOp}(y,y'),$$

and note that ϕ_F has at most 3 variables. □

However, it is clear that the converse does not hold: there are first-order logic formulas using only three variables that have no equivalent in FOXPath. This is because FOXPath gives no added expressiveness on the navigational structure of a document. Formally, we say that a Boolean query Q over XML documents is *navigational* if Q cannot distinguish two documents that are isomorphic as unranked ordered trees (that is, the two documents have isomorphic interpretations for σ_{nav}). Then we have

Proposition 3.15. *Any navigational Boolean query expressible in* FOXPath *is expressible in* NavXPath*, and hence is expressible in FO^2. In particular (by [Etessami et al. 2002]), there are $FO[\sigma_{nav}, R_{\mathsf{descendant}}]$ queries not expressible in* FOXPath.

Proof Sketch. We say that a set of XML documents R is a *representative family* iff for each XML-tree t there is an XML document d such that d is an expansion of t and $d \in R$ (i.e. the reduct-map is surjective).

Let ϕ be an arbitrary FOXPath query that is navigational.

Perform the following rewriting of ϕ. Replace each atomic filter of form $\pi/@a = \pi'/@b$ or $\pi/@a \leq \pi'/@b$ by $\pi \wedge \pi'$ and each atomic filter of form $\pi/@a \neq \pi'/@b$ or $\pi/@a < \pi'/@b$ by `false`. Call the NavXPath query obtained by this rewriting ϕ'. It is easy to observe that for any labeled tree t, it is true for the expansion to the XML document d obtained by mapping each node to the same value, say $val : x \mapsto 1$ for all x, that $\phi(t) \equiv \phi'(d)$. Thus the set of these expansions is a representative family, and for all navigational queries ϕ and all d from that representative family, $\phi(d) \equiv \phi'(d)$. The theorem then follows from the following

Claim 3.16. *If ϕ and ϕ' are navigational queries and $\phi(d) \equiv \phi'(d)$ for all XML documents d in a representative family, then $\phi(d') \equiv \phi'(d')$ on all XML documents d'.*

Proof of claim: Assume that there exists a representative family R such that $\phi(d) \equiv \phi'(d)$ for all $d \in R$. Given an arbitrary XML documents d, we take its reduct d_0 to σ_{nav}. Of course there exists an expansion $d_R \in R$ of d_0. By assumption,

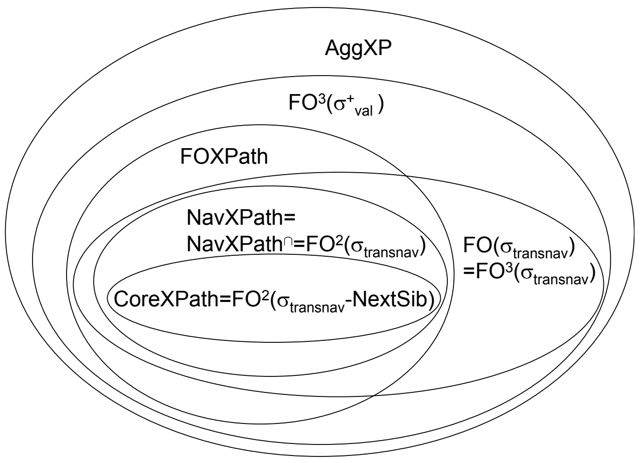


Fig. 2. Expressive power of XPath Language fragments versus first-order languages.

$\phi(d_R) \equiv \phi'(d_R)$. If ϕ and ϕ' are navigational, $\phi(d_R) \equiv \phi(d)$ and $\phi'(d_R) \equiv \phi'(d)$. Thus $\phi(d) = \phi'(d)$.

In the case of AggXPath, in contrast, it is known that all navigational first-order queries are expressible:

PROPOSITION 3.17. *Any $FO[\sigma_{transnav}]$ boolean query is expressible in* AggXPath.

In particular, the axis immediately-following is expressible in AggXPath.

Proof Sketch. We use a result of [Marx 2004a], which states that it is sufficient to show closure under the following variant of the modal until operators. For an axis $\alpha \in \{$child, parent, next-sibling, previous-sibling$\}$, we write α^+ for the corresponding transitive axis (child$^+$ = descendant , etc.) and α^* for the union of α^+ with the self axis (child$^+$ = descendant-or-self, etc.). For axis $\alpha \in \{$child, parent, next-sibling, previous-sibling$\}$ and queries $Q_1(x), Q_2(x)$, the query $\text{Until}_\alpha(Q_2, Q_1)(x)$ ("property Q_1 until property Q_2") holds at a node n iff there is n' such that $R_{\alpha^+}(n, n')$ holds, $Q_2(n')$ holds, and for all n'' such that $R_{\alpha^+}(n, n'')$ and $R_{\alpha^+}(n'', n')$ we have $Q_1(n'')$. Marx has shown (combination of Theorems 6 and 7 of [Marx 2004a]) that any language containing unary label tests and closed under boolean operations and the until operators above can express any first-order formula in one free variable. Since AggXPath is closed under boolean operations, it is thus sufficient to show closure under until. But if E_1 and E_2 are AggXPath expressions returning Booleans, then $\text{Until}_\alpha(E_2, E_1)$ can be expressed as $\alpha^+::*[E_2] \wedge \neg\big(\text{count}(\alpha^+::*[\neg E_1]/\alpha^+::*[E_2]) = \text{count}(\alpha^+::*[E_2])\big)$. □

A summary of our expressiveness results is shown in Figure 2.

3.4 Further Bibliographic Remarks

In this section, we have discussed exact characterizations of sublanguages of XPath via logic and tree patterns. We have focused on the relationship between NavXPath and logics, because this is where the cleanest characterization can be shown. However, the relationship between XPath 1.0 and logics with few variables extends to logics that manipulate data, as shown in our results on FOXPath above. This relationship will play a role in the complexity results of the next section. The relationship between PNavXPath queries and acyclic first-order queries is explored further in [Gottlob et al. 2004].

There are other formalisms in which NavXPath and CoreXPath can be embedded as a strict subset, and we review them below.

[Neven and Schwentick 2002] deals with *query automata*, an automata model that defines *NodeSet* queries. Query automata have the expressiveness of Monadic

Second Order Logic, hence they are strictly more powerful than NavXPath. [Frick et al. 2003; Koch 2003] deal with a variant of non-deterministic tree automata that can define unary rather than Boolean queries. [Carme et al. 2004] define queries on unranked trees via automata that work on binary encodings. As with query automata, both these formalisms strictly subsume NavXPath in expressiveness. One starting point in looking for an automata characterization of XPath is [Schwentick et al. 2001], which gives a characterization of two-variable logic over strings in terms of partially-ordered two-way deterministic automata. We do not know of a similar characterization for two-variable logic on trees. A comprehensive survey of the relationship of XML queries to automata is given in [Schwentick 2007].

As mentioned in the introduction, there is a natural connection between navigational XPath and modal logics, which was first observed in [Miklau and Suciu 2002] and [Gottlob and Koch 2002] and subsequently revisited in several works (e.g. [Marx 2004b; 2004a; Afanasiev et al. 2004]). The closest relation is to linear temporal logic (LTL) and Propositional Dynamic Logic (PDL). LTL formulas give properties of nodes within a string. They are built up from formulas checking the label of a node via boolean operators and the operators "at the next place ϕ" "eventually ϕ" and "ϕ until ψ". The restriction of LTL obtained by removing the until operator is called Unary Temporal Logic. NavXPath qualifiers can be considered as an extension of Unary Temporal Logic from strings to trees. In particular, the expressiveness of NavXPath qualifiers over strings is exactly that of Unary Temporal Logic. Branching time temporal logics, such as CTL^*, generalize LTL from strings to graphs, rather than to trees. The techniques for proving expressiveness results for NavXPath qualifiers borrow heavily from the prior work on LTL and CTL^* expressiveness.

PDL formulas give formulas mapping nodes to nodesets within an edge-labeled graph. They are built up from operators that can move forward on any labeled edge. XPath nodeset expressions can be considered, roughly as PDL formulas where the edge-labeled graph is obtained from an ordered tree. Many of the static analysis results (see, for example, Theorem 5.8) follow from modifying prior results for PDL.

We do not pursue the relationship with either automata or modal logics in detail because the expressiveness of XPath does not *exactly* match either PDL or LTL. An approach to filling this gap would be to define natural extensions of either temporal logic or PDL to deal with trees. For temporal logics, see [Barcelo and Libkin 2005] for an extended discussion of this approach, while for PDL see [Afanasiev et al. 2005].

A natural question is what should be added to NavXPath to capture all of first-order logic. It is known that first-order logic with 3 variables captures FO (established in [Marx 2004a] for ordered unranked trees). Marx [Marx 2004a] proposes two extensions of NavXPath to capture FO^3, and thus be first-order complete. One is by adding a path complementation feature to NavXPath and the other is by introducing conditional axes in the spirit of the *until* operator of CTL. These results can be seen as extensions of Kamp's Theorem [Kamp 1968], which states that linear temporal logic (with "until") captures first-order logic over infinite words, to the setting of unranked trees.

4. COMPLEXITY AND EFFICIENT EVALUATION

This section studies the complexity of XPath queries. XPath is a variable-free query language in which many queries – in particular, all NavXPath queries – are tree-shaped in a natural sense when converted into first-order logic. At the same time the navigational structure of XML documents is tree-shaped. We first look at some of the classical results about tree-like queries and queries on tree-like structures. Then we explore the connections between the powerful notion of hypertree-width and XPath and show the new result that conjunctive FOXPath queries have hypertree-width 2. After that, we generalize from XPath evaluation based on hypertree decompositions and illustrate the dynamic programming technique that has yielded a polynomial time algorithm for full XPath 1.0. Then we survey the parallel complexity of XPath and give a new simplified proof that XPath is hard for polynomial time. Finally, we study XPath processing on data streams and give an overview over further work on efficient XPath processing.

4.1 Complexity Background

Throughout this section, we will consider logics and query languages as problem classes and will simply identify the languages with their evaluation problems. Two kinds of complexity of query evaluation will be considered, *data complexity* (where queries are assumed to be fixed and data variable) and *combined complexity* (where both data and query are considered variable) [Vardi 1982].

We briefly discuss the complexity classes and some of their characterizations used throughout the remainder of this survey. For more thorough surveys of complexity classes and the related theory see [Johnson 1990; Papadimitriou 1994; Greenlaw et al. 1995].

By PTime, ExpTime, NExpTime, LogSpace, NLogSpace, and PSpace we denote the well-known complexity classes of problems solvable on Turing machines in deterministic polynomial time, deterministic exponential time, nondeterministic exponential time, deterministic logarithmic space, nondeterministic logarithmic space, and (deterministic) polynomial space, respectively. By NP, we denote the decision problems solvable in nondeterministic polynomial time and co-NP denotes the class of their complements.

It is a widely-held conjecture that problems complete for PTime are inherently sequential and cannot profit from parallel computation (cf. e.g. [Greenlaw et al. 1995]). Instead, a problem is called *highly parallelizable* if it can be solved within the complexity class NC of all problems solvable in polylogarithmic time on a polynomial number of processors working in parallel [Greenlaw et al. 1995].

A simple model of parallel computation is that of Boolean circuits. By a monotone circuit, we denote a circuit in which only the input gates may possibly be negated. All other gates are either $\wedge$-gates or $\vee$-gates (but no $\neg$-gates). A family of circuits is a sequence $\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2, \ldots$, where the n-th circuit $\mathcal{G}_n$ has n inputs. Such a family is called LogSpace-*uniform* if there exists a LogSpace-bounded deterministic Turing machine which, on the input of n bits 1 (the string 1^n), outputs the circuit $\mathcal{G}_n$. A family of circuits has *bounded fan-in* if all of the gates in these circuits have fan-in bounded by some constant. On the other hand, a family of monotone circuits is called *semi-unbounded* if all $\wedge$-gates are of bounded fan-in (without loss of generality, we may restrict the fan-in to two) but the $\vee$-gates may have unbounded fan-in.

NC^i denotes the class of languages recognizable using LogSpace-uniform Boolean circuit families of polynomial size and depth $O(\log^i n)$ (in terms of the size n of the input). SAC^1 is the class of languages recognizable by LogSpace-uniform families of semi-unbounded circuits of depth $O(\log n)$ (SAC^1 circuits).

A nondeterministic auxiliary pushdown automaton (NAuxPDA) is a nondeterministic Turing machine with a distinguished input tape, a worktape, and a stack (of which strictly only the topmost element can be accessed at any time).

LogCFL is usually defined as the complexity class consisting of all problems LogSpace-reducible to a context-free language. There are two important alternative characterizations of LogCFL that we are going to use. They are recalled in Proposition 4.1 and 4.2, respectively.

Proposition 4.1 [Venkateswaran 1991]. LogCFL $= \mathrm{SAC}^1$. SAC^1 *Circuit Value is* LogCFL*-complete.*

PROPOSITION 4.2 [SUDBOROUGH 1977]. LOGCFL *is the class of all decision problems solvable by a NAuxPDA with a logarithmic space-bounded worktape in polynomial time.*

We have LOGSPACE $\subseteq$ NLOGSPACE $\subseteq$ LOGCFL $\subseteq$ NC^2 $\subseteq$ NC $\subseteq$ PTIME $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME $\subseteq$ NEXPTIME. All inclusions $\subseteq$ are suspected to be strict, and all these complexity classes are closed under LOGSPACE-reductions.

Unless stated otherwise, we assume the input represented as a σ_{dom}-structure encoded in the usual way.

4.2 Tree-like Data and Tree-like Queries

As a warm-up, we use the well-studied graph-theoretical notion of *tree-width* to derive a few results about the complexity of XPath that follow immediately from the literature.

Let $G = (V^G, E^G)$ be a graph. A *tree decomposition* of G is a pair (T, χ) such that T is a rooted tree with nodes V^T, χ is a function $\chi : V^T \rightarrow 2^{V^G}$ that maps each node of tree T to a subset of V^G, for each edge $(u, v) \in E^G$ there exists a node $w \in V^T$ such that $u, v \in \chi(w)$, and for each node $u \in V^G$, the set $\{v \in V^T \mid u \in \chi(v)\}$ induces a connected subtree of T. The *width* of tree decomposition (T, χ) is defined as $\big(\max\{|\chi(v)| \mid v \in V^T\}\big) - 1$. The *tree-width* of a graph G is the smallest width over all tree decompositions of G. Intuitively, graphs of low tree-width are very tree-like. As a special case, the connected graphs of tree-width one are precisely the trees. An example of a graph and a tree decomposition (of width 2) for it is given in Figures 3 (a) and (b), respectively.

We say that a structure consisting only of unary and binary relations has tree-width k if the union of (the symmetric closure of) its binary relations has tree-width k. We do not give a formal definition of the general case of *queries of bounded tree-width* here; however, for conjunctive queries Q over a vocabulary of at most binary relation symbols, the *tree-width of Q* is defined as the tree-width of the graph $G = (V, E)$ where V consists of the variables of Q and $(x, y), (y, x) \in E$ if there is an atom $a(x, y)$ in Q.

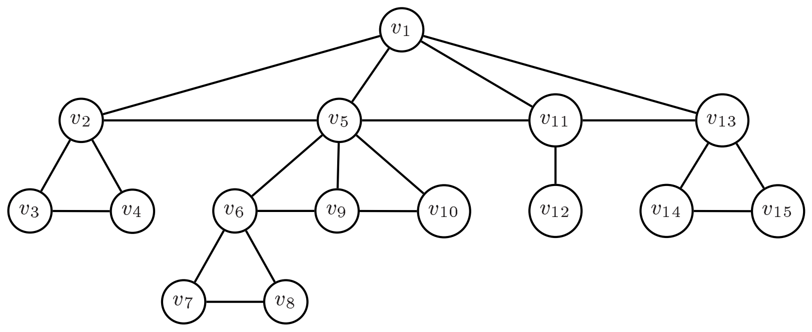
§*1: Tree-like data lead to linear-time data complexity.* The Boolean MSO queries on trees labeled with a finite alphabet (e.g. σ_{nav}-trees) define precisely the *regular tree languages*, which correspond to the *deterministic bottom-up tree automata* [Thatcher and Wright 1968; Doner 1970; Brüggemann-Klein et al. 2001]. Each Boolean MSO query can be mapped to such an automaton, whose acceptance of a given input tree can be checked in linear time in the size of the tree (traversing it once bottom-up). Thus, Boolean MSO queries on trees have linear-time data complexity. A slightly more general version of this fact for bounded tree-width structures is known as Courcelle's Theorem [Courcelle 1990], which can be further generalized to

THEOREM 4.3 [FLUM ET AL. 2002]. *Let* $\mathbf{C}$ *be a class of structures of bounded tree-width. For a fixed MSO formula* ϕ*, there is an algorithm that evaluates* ϕ *on each structure* $\mathcal{A} \in \mathbf{C}$ *in time* $O(|\mathcal{A}| + |\phi(\mathcal{A})|)$.

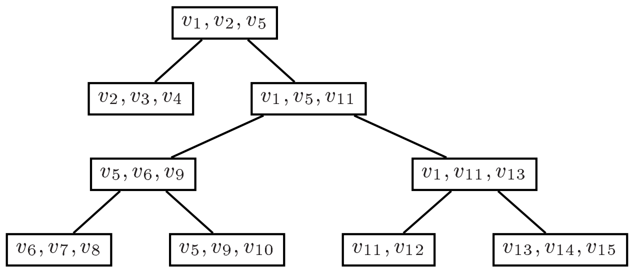
That is, this algorithm runs in time linear in the size of the input and the output, and in particular in linear time *in the size of the input* on MSO formulas with at most one free variable.

It can be verified that unranked ordered trees represented by σ_{nav}-structures, that is, the union of their binary relations R_{child} and $R_{\mathsf{next\text{-}sibling}}$, have tree-width two[2]

[2]Note, however, that in the context of MSO, it is more wide-spread [Neven 2002; Gottlob and Koch 2004] to use a signature σ'_{nav} obtained from σ_{nav} by replacing R_{child} by a relation $\mathsf{FirstChild}$ such that $\mathsf{FirstChild}(x, y)$ iff y is the *leftmost* child of x. Then, MSO on σ_{nav} and σ'_{nav} are equivalent and all σ'_{nav}-structures have tree-width 1.



(a)



(b)

Fig. 3. A σ_{nav}-tree is a graph of tree-width two.

(see Figure 3, where each node v is labeled with $\chi(v)$). Transitive axis relations such as $R_{\mathsf{descendant}}$ or $R_{\mathsf{following\text{-}sibling}}$ (cf. Section 2.1) do not have bounded tree-width in general, but it is not difficult to map NavXPath queries with transitive axes to MSO over signature σ_{nav} [Gottlob and Koch 2002]. The construction is similar to the one of Theorem 3.2 mapping NavXPath to FO^2, defining $R^*(x, y)$, where R^* is the reflexive and transitive closure of relation R, in MSO as $\forall S \ \big(S(x) \wedge \forall u \forall v \ S(u) \wedge R(u, v) \rightarrow S(v)\big) \rightarrow S(y)$. From this we can conclude the following bound.

COROLLARY 4.4. NavXPath *NodeSet queries (and hence,* CoreXPath *NodeSet queries) are in linear time with respect to data complexity.*

§*2: Tree-like data do not yield low combined complexity.* The usual technique for proving linear-time data complexity of MSO is by reduction to automata. For unary MSO formulas, somewhat sophisticated automata with a capability for selecting nodes are required. It has been observed that such automata with the power of unary MSO can be designed to traverse the data tree only twice [Neven and Van den Bussche 2002; Frick et al. 2003]. Reductions from MSO to automata do not yield good upper bounds on the combined complexity of NavXPath, however. Indeed, they are necessarily nonelementary [Meyer 1975; Reinhardt 2002] (i.e., their cost cannot be bounded by any tower of exponentials $2^{2^{2^{\cdot^{\cdot^{2^n}}}}}$ of fixed height). For NavXPath, a doubly exponential translation to *selecting tree automata* [Frick et al. 2003] is implicit in [Koch 2003].

§*3: Tree-like queries yield polynomial-time combined complexity.* While MSO over trees is known to be PSPACE-complete with respect to combined complexity, FO^k (even over arbitrary relational structures) is known to be in time $O(n^k * |Q|)$: [3]

PROPOSITION 4.5 [KOLAITIS AND VARDI 2000]. *Conjunctive FO^{k+1} queries have tree-width $\leq k$.*

[3]This can be shown directly without tree-width as well [Vardi 1995], however.

THEOREM 4.6 [CHEKURI AND RAJARAMAN 1997]. *Given a Boolean conjunctive query Q of tree-width k and a database $\mathcal{A}$ with domain size n, Q can be evaluated on the database in time $O((n^{k+1} + |\mathcal{A}|) * |Q|)$.*

Both results generalize from conjunctive to FO queries [Flum et al. 2002].

Since boolean NavXPath queries can be translated efficiently, in linear time, into equivalent FO^2 queries (Theorem 3.2) and FOXPath queries can be translated in linear time into FO^3 (Proposition 3.14),

COROLLARY 4.7. *Boolean* NavXPath *and* FOXPath *can be evaluated in time $O(|\mathcal{D}|^2 \cdot |Q|)$ and $O(|\mathcal{D}|^3 \cdot |Q|)$, respectively, on a σ_{dom} structure $\mathcal{D}$.*

As we will see later on in this section, these combined complexity bounds can be improved upon.

4.3 Hypertree-width and Conjunctive XPath

All results of Sections 4.3 and 4.4 will apply both to nodeset and to Boolean queries of the respective fragments indicated.

Let Q be a conjunctive query over a relational database, and let $vars(Q)$, $free(Q)$, and $atoms(Q)$ denote the set of variables, free variables, and atoms occurring in Q, respectively.

A (complete) *hypertree decomposition* of Q is a triple (T, χ, λ) such that T is a rooted tree with nodes $V(T)$ and root node r, $\chi : V(T) \to 2^{vars(Q)}$ maps each node of tree T to a set of variables from Q, $\lambda : V(T) \to 2^{atoms(Q)}$ maps each node of T to a set of body atoms of Q,

(1) $free(Q) \subseteq \chi(r)$,
(2) for each atom $A \in atoms(Q)$, there exists a node $v \in V(T)$ such that $A \in \lambda(v)$ and $vars(A) \subseteq \chi(v)$,
(3) for each variable $x \in vars(Q)$, the set $\{v \in V(T) \mid x \in \chi(v)\}$ induces a connected subtree of T, and
(4) for each node $v \in V(T)$, $\chi(v) \subseteq vars(\lambda(v))$ and

$$vars(\lambda(v)) \cap \bigcup\{\chi(v') \mid v = v' \text{ or } v' \text{ is a descendant of } v \text{ in } T\} \subseteq \chi(v).$$

The *width* of a hypertree decomposition (T, χ, λ) is the maximum number of atoms occurring in any single node of T, i.e. $\max\{|\lambda(v)| \mid v \in V(T)\}$. The *hypertree-width* of a conjunctive query Q is the smallest width over all hypertree decompositions of Q. The conjunctive queries of hypertree-width 1 coincide with the so-called *acyclic* conjunctive queries (cf. e.g. [Abiteboul et al. 1995]). As shown in [Yannakakis 1981], the acyclic conjunctive queries can be evaluated in time $O(n \cdot |Q|)$. Yannakakis' result was generalized to hypertree-width k, for arbitrary k:

THEOREM 4.8 [GOTTLOB ET AL. 2002]. *Let Q be a conjunctive query and $\mathcal{H}$ a hypertree decomposition of width k of Q. Then Q can be evaluated on a database $\mathcal{A}$ in time $O((|\mathcal{H}| + |\mathcal{A}|)^k)$.*

Let σ'_{dom} be the signature obtained from σ_{dom} by replacing each attribute function $@A$ by its graph (i.e., the binary relation $\{(n, @A(n)) \mid n \in \mathit{Node}\}$) and adding the relations $R_{\mathsf{descendant}}$ and $R_{\mathsf{following\text{-}sibling}}$.

A considerable fragment of FOXPath can be modeled by conjunctive queries over a structure of relational signature σ'_{dom}. We say that a FOXPath query (resp., NavXPath query) is *conjunctive* (and connected) if it does not use disjunction, negation, inequalities (i.e., expressions $p\,\mathsf{RelOp}\,p'$ with $\mathsf{RelOp} \neq$ "="), or the root slash /. The notions of hypertree decomposition and hypertree-width can be readily applied to conjunctive FOXPath (and thus NavXPath) queries. A conjunctive FOXPath query maps to a conjunctive query over σ'_{dom}, and we can speak of its hypertreewidth using this mapping.

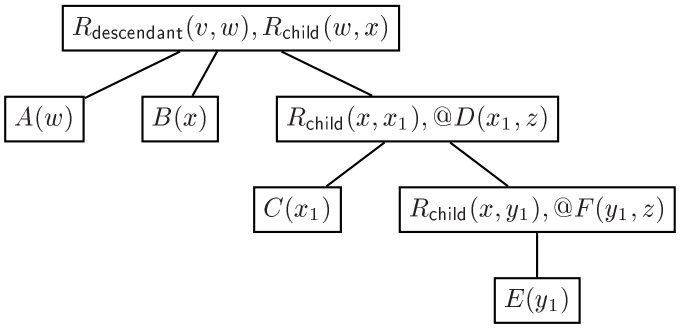
Example 4.9. The conjunctive FOXPath query

$$\mathsf{descendant}::A/\mathsf{child}::B[\mathsf{child}::C/@D = \mathsf{child}::E/@F]$$

can be phrased as a conjunctive query over signature σ'_{dom}

$$Q(v, x) \leftarrow R_{\mathsf{descendant}}(v, w), A(w), R_{\mathsf{child}}(w, x), B(x), R_{\mathsf{child}}(x, x_1), C(x_1), @D(x_1, z), \\ R_{\mathsf{child}}(x, y_1), E(y_1), @F(y_1, z).$$

Consider the following hypertree decomposition, $\mathcal{H}$, of Q, where the nodes v have been labeled with $\lambda(v)$ and $\chi(v) = vars(\lambda(v))$:



$R_{\mathsf{descendant}}(v, w), R_{\mathsf{child}}(w, x)$
$A(w)$
$B(x)$
$R_{\mathsf{child}}(x, x_1), @D(x_1, z)$
$C(x_1)$
$R_{\mathsf{child}}(x, y_1), @F(y_1, z)$
$E(y_1)$

Note that $\mathcal{H}$ is of width 2. There exists obviously no hypertree decomposition of width 1: the atoms $\{R_{\mathsf{child}}(x, x_1), @D(x_1, z), R_{\mathsf{child}}(x, y_1), @F(y_1, z)\}$ of Q induce a cycle. Thus Q is of hypertree-width 2. $\square$

By Propositions 4.5 and 3.14, conjunctive FOXPath queries have tree-width ≤ 2. It is known that conjunctive queries of tree-width k have hypertree-width $\leq k + 1$ [Gottlob et al. 2002], so we can obtain the $O(n^3)$ data complexity bound observed in Corollary 4.7 also from Theorem 4.8. However, fortunately,

Theorem 4.10. *The conjunctive* FOXPath *NodeSet queries have hypertree-width* ≤ 2.

Proof. We first compute a first-order query (using just $\exists$ and $\wedge$) over σ'_{dom} for a given conjunctive FOXPath query and then show that it yields a hypertree decomposition of width ≤ 2. From the first-order formula an equivalent relational algebra plan can be obtained immediately by rewriting $\wedge$ by a join and $\exists$ by a projection

We will assume that our query is a path expression p. The proof works analogously for qualifiers. We translate p into a first-order formula $FO(p)_2$ as follows:

$$\begin{aligned}
FO(axis)_2(x, y) &:= R_{axis}(x, y) \\
FO(step[q])_2(x, y) &:= FO(step)_2(x, y) \wedge FO(q)_1(y) \\
FO(p/step)_2(x, z) &:= \exists y\ FO(p)_2(x, y) \wedge FO(step)_2(y, z) \\
FO(\mathsf{lab}() = L)_1(x) &:= L(x) \\
FO(p)_1(x) &:= \exists y\ FO(p)_2(x, y) \\
FO(q \wedge q')_1(x) &:= FO(q)_1(x) \wedge FO(q')_1(x) \\
FO(p/@A = p'/@B)_1(x) &:= \exists z\ \big(\exists y_1\ FO(p)_2(x, y_1) \wedge @A(y_1, z)\big)\ \wedge \\
&\qquad \big(\exists y_2\ FO(p')_2(x, y_2) \wedge @B(y_2, z)\big)
\end{aligned}$$

Without loss of generality, we will assume that there are no two distinct occurrences of existential quantification over the same variable in $FO(p)_2$; thus, any two occurrences of the same variable name in formula $FO(p)_2$ indeed refer to the same variable.

$FO(\cdot)_2$ is only a minor variation of $[\![\cdot]\!]_{NodeSet}$ and it is easy to verify that $FO(p)_2$ defines a binary relation $\{(n, n') \mid n' \in [\![p]\!]_{NodeSet}(n)\}$.

We now construct a hypertree decomposition of $FO(p)_2$. Consider the parse tree T of formula $FO(p)_2$. This parse tree has relation atoms as its leaves and $\exists x$- and $\wedge$-labels on its internal nodes. Each node of the tree corresponds to a subformula ϕ of $FO(p)_2$. We will identify each tree node with the subformula ϕ it denotes.

We define a function λ that maps each node ϕ of T to a set of leaf nodes (and thus relational atoms). We do this inductively, bottom-up:

(i) for each leaf node ϕ, $\lambda(\phi) := \{\phi\}$;

(ii) for each node ϕ of the form $\psi_1(x) \wedge \psi_2(x)$, $\psi_1(x,y) \wedge \psi_2(y)$, or $\psi_1(x,y) \wedge \psi_2(x,y)$, let $\lambda(\phi) := \lambda(\psi_1)$;

(iii) for each node $\phi = \psi_1(x,y) \wedge \psi_2(y,z)$, let $\lambda(\phi) := \{\psi'\} \cup \lambda(\psi_2)$, where ψ' is any atom over x from $\lambda(\psi_1)$; finally,

(iv) for each node $\phi = \exists x\, \psi$, $\lambda(\phi) := \lambda(\psi)$.

Note, in particular, that each free variable of ϕ occurs in at least one atom of $\lambda(\phi)$. Now let function χ map each node ϕ of T to $vars(\lambda(\phi))$.

To verify that (T, χ, λ) is indeed a hypertree decomposition of p, we have to check points (1) to (4) of the definition. (1) and (4) are due to the definition of χ as $\phi \mapsto vars(\lambda(\phi))$. (2) is immediate from (i). The connectedness condition (3) follows from the fact that in a first-order query without any two distinct occurrences of existential quantification over the same variable, the nodes of parse tree T that have x as a free variable plus the node $\exists x\, \psi$ if x is not free in the query induce a connected subtree of T.

Let us now consider the sizes $|\lambda(\phi)|$ for all nodes ϕ of T. The most interesting case is $\phi = \psi_1(x,y) \wedge \psi_2(y,z)$. Observe that in this case ψ_2 is either a step expression or a leaf, and thus $|\lambda(\psi_2)| = 1$, so $|\lambda(\phi)| = 2$. It can be shown by a straightforward induction that for all nodes ϕ, $|\lambda(\phi)| \leq 2$, so our query has hypertree-width ≤ 2. □

This result by construction of course holds for nodeset queries and thus also for Boolean queries.

EXAMPLE 4.11. For the query of Example 4.9,

$$FO(\mathsf{descendant}::A/\mathsf{child}::B[\mathsf{child}::C/@D = \mathsf{child}::E/@F])_2(v,x)$$

evaluates to the first-order formula

$$\begin{aligned}\exists w\ (R_{\mathsf{descendant}}(v,w) \wedge A(w)) \wedge \big(R_{\mathsf{child}}(w,x) \wedge \big(B(x) \wedge \\ \exists z\ (\exists x_1\ (R_{\mathsf{child}}(x,x_1) \wedge C(x_1) \wedge @D(x_1,z))) \wedge \\ (\exists y_1\ (R_{\mathsf{child}}(x,y_1) \wedge E(y_1) \wedge @F(y_1,z))))\big)\big)\end{aligned}$$

the parse tree of which is shown in Figure 4. The leaf nodes in the figure have been labeled $l_1, l_2, l_3, \ldots$ from left to right and the interior nodes ϕ of the parse tree of the formula have been annotated with $\lambda(\phi)$. Again, $\chi(\phi) = vars(\lambda(\phi))$. This yields the hypertree decomposition constructed in the proof. □

The transformation of the previous proof can be implemented so as to compute both first-order query and hypertree decomposition in linear time. By the latter observation and Theorem 4.8 we thus see that Conjunctive $\mathsf{FOXPath}$ can be evaluated in time $O((|Q| + |\mathcal{D}|)^2)$.

We give a direct proof of the following (close but incomparable) bound.

PROPOSITION 4.12. *Conjunctive* $\mathsf{FOXPath}$ *NodeSet queries can be evaluated on* σ'_{dom}*-structures* $\mathcal{D}$ *in time* $O(|Q| \cdot |\mathcal{D}|^2)$.

Proof. Let us now consider relational algebra queries $ALG(p)$ and $ALG(q)$ corresponding to the first-order (calculus) queries $FO(p)_2$ and $FO(q)_1$ of the previous

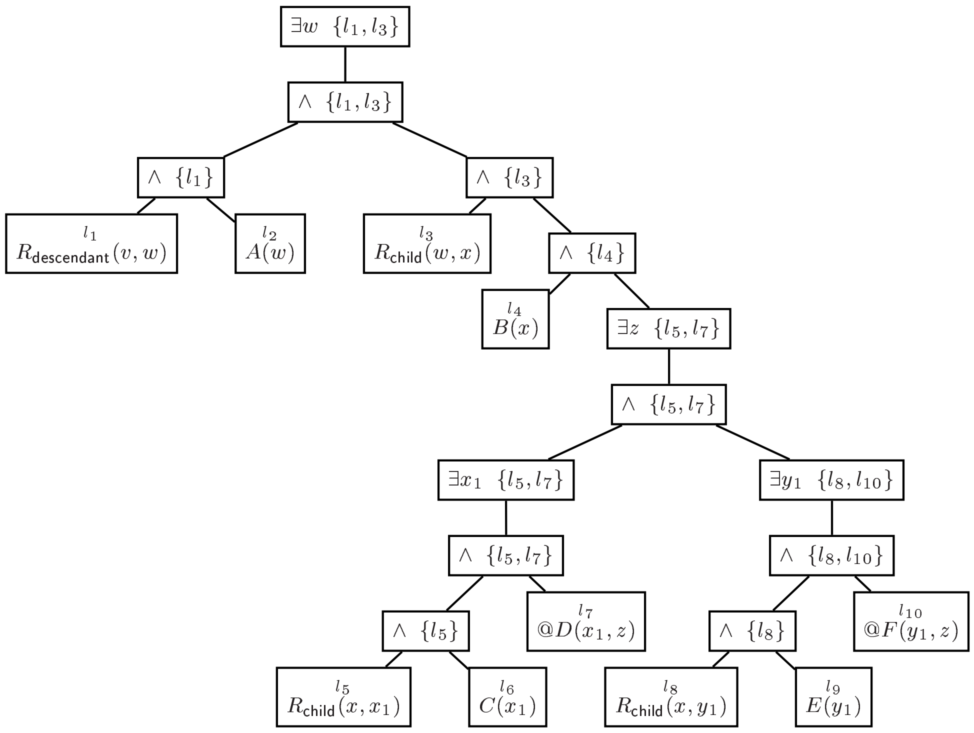

$\exists w\ \{l_1, l_3\}$
$\wedge\ \{l_1, l_3\}$
$\wedge\ \{l_1\}$
$\wedge\ \{l_3\}$
l_1 $R_{\mathsf{descendant}}(v, w)$
l_2 $A(w)$
l_3 $R_{\mathsf{child}}(w, x)$
$\wedge\ \{l_4\}$
l_4 $B(x)$
$\exists z\ \{l_5, l_7\}$
$\wedge\ \{l_5, l_7\}$
$\exists x_1\ \{l_5, l_7\}$
$\exists y_1\ \{l_8, l_{10}\}$
$\wedge\ \{l_5, l_7\}$
$\wedge\ \{l_8, l_{10}\}$
$\wedge\ \{l_5\}$
l_7 $@D(x_1, z)$
$\wedge\ \{l_8\}$
l_{10} $@F(y_1, z)$
l_5 $R_{\mathsf{child}}(x, x_1)$
l_6 $C(x_1)$
l_8 $R_{\mathsf{child}}(x, y_1)$
l_9 $E(y_1)$

Fig. 4. Hypertree decomposition of the query of Example 4.9 as constructed in the proof of Theorem 4.10.

proof. The translation is standard [Abiteboul et al. 1995] and just requires rewriting existential quantification by projection and conjunction by join.

As with the subformulas of ϕ in $FO(p)_2$, each subexpression of $ALG(p)$ defines a relation that is a subset of the product of at most two base relations $\lambda(\phi)$, and is thus of size at most $O(|\mathcal{D}|^2)$.

Query evaluation requires no more than $|Q|$ relational algebra operations (projections or joins). The projections $\pi_{\vec{A}}R$ are obviously operations that run in time linear in $|R|$. Joins *guarded* by one of the input relations (corresponding to formulae $\psi_1(x,y) \wedge \psi_2(x,y)$, $\psi_1(x,y) \wedge \psi_2(y)$, and $\psi_1(y) \wedge \psi_2(y)$) can be evaluated in time linear in the sum of the sizes of the two relations joined by first building a bitfield for testing whether tuples are true in ψ_2 and then using it to filter the tuples of ψ_1.

The most interesting case is a join corresponding to formula $\psi_1(x,y) \wedge \psi_2(y,z)$. Let $[\![\phi]\!]$ be the relation defined by first-order formula ϕ. We first compute the relations $R_1^y = \{x \mid \psi_1(x,y)\}$, for each y such that $\exists z\ \psi_2(y,z)$, in total time $O(|[\![\psi_1]\!]| + |[\![\psi_2]\!]|)$. Then we compute our join as the union of the sets $\{(x,y,z) \mid R_1^y(x)\}$, for each tuple $\psi_2(y,z)$. As mentioned in the previous proof, ψ_2 always defines a subset of an input relation, so this union can be formed in time $O(|\mathcal{D}| \cdot |[\![\psi_2]\!]|) = O(|\mathcal{D}|^2)$. □

Conjunctive NavXPath queries are acyclic (see [Gottlob et al. 2005]) and can therefore be evaluated using Yannakakis' algorithm (or by precisely the techniques from the previous two proofs) both in linear time in the data *and* efficiently in the size of the query.

PROPOSITION 4.13. *Conjunctive* NavXPath *NodeSet queries can be evaluated in time $O(|\mathcal{D}| \cdot |Q|)$ on $(\sigma_{nav}, R_{\mathsf{descendant}}, R_{\mathsf{following\text{-}sibling}})$-structures $\mathcal{D}$.*

4.4 Beyond Conjunctive Queries

The conjunctive query processing techniques based on hypertree decompositions of the previous section leave three features of FOXPath unaddressed:

(1) Conjunctive FOXPath excludes disjunction, union, negation, inequalities, and disconnected queries (via the root / in conditions).

(2) We assumed that the data tree is given by σ^+_{val}-structures, which include binary relations for transitive axes such as descendant. If we assume transitive axis relations present in the structure $\mathcal{D}$ representing a tree with domain A and therefore $|\mathcal{D}| = O(|A|^2)$, our upper bound on time of $O(|\mathcal{D}|^2 \cdot |Q|)$ from Proposition 4.12 deteriorates to time $O(|\mathcal{D}|^4 \cdot |Q|)$ when the input structure $\mathcal{D}$ is now in σ_{dom}.

(3) Finally, we did not deal with inequalities $\mathsf{RelOp} \in \{\neq, <, \leq\}$ in expressions $e\,\mathsf{RelOp}\,e'$.

The following result deals with all these issues.

THEOREM 4.14. *A* FOXPath *NodeSet query Q can be evaluated on σ_{dom}-structures with domain A in time $O(|A|^2 * |Q|)$.*

Proof.

(1) We complete the mapping ALG of the previous proof by the operations of FOXPath missing from conjunctive FOXPath:
—$ALG(p \mid p') := ALG(p) \cup ALG(p')$
—$ALG(q \vee q') := ALG(q) \cup ALG(q')$
—$ALG(\neg q) := A - ALG(q)$

(2) Next we would like to eliminate transitive axis relations such as descendant from the signature.
[Gottlob et al. 2005] gives algorithms for computing, given a set S of tree nodes and any XPath axis α, the set of nodes

$$\alpha(S) = \{y \mid x \in S \wedge R_\alpha(x, y)\}$$

in time $O(|Node|)$. Consider the unary operations

$$\bowtie_{\alpha[q]}: R \mapsto \{(x, z) \mid \exists y\, R(x, y) \wedge R_\alpha(y, z) \wedge [\![q]\!]_{Boolean}(z)\},$$

which can be evaluated in quadratic time by first partitioning R into sets $S_x = \{y \mid R(x, y)\}$, for each x, and then computing the union over x of the sets $\{(x, y) \mid y \in \alpha(S_x) \wedge [\![q]\!]_{Boolean}(y)\}$.
Now we can evaluate $[\![p/\alpha[q_1] \ldots [q_n]]\!]$ as $\alpha[q_1 \wedge \cdots \wedge q_n]([\![p]\!])$ in quadratic time, for any axis α, even if our structure is just of signature σ_{dom}.

(3) Let α^{-1} denote the inverse of axis α (i.e., $R_{\alpha^{-1}}$ is the inverse of R_α). To compute a query plan for an inequality

$$\alpha_1[q_1]/\alpha_2[q_2]/\cdots/\alpha_n[q_n]/@A\ \mathsf{RelOp}\ \beta_1[q_1']/\beta_2[q_2']/\cdots/\beta_n[q_n']/@B$$

with $\mathsf{RelOp} \neq$ "=", we first compute the binary relation $\mathsf{RelOp}_{@A,@B}$ (see the definition of σ^+_{val} in Section 3.3) in time $O(|A|^2)$. Using the fact that the joins above can be computed in quadratic time, we see that we can compute the following relation S in quadratic time $|A|^2$ times the size of S:

$$S := \bowtie_{\beta_1^{-1}} (\bowtie_{\beta_2^{-1}[q_1']} (\bowtie_{\beta_3^{-1}[q_2']} (\cdots \bowtie_{\beta_n^{-1}[q_{n-1}']} (\bowtie_{\mathsf{self}[q_n']} (\mathsf{RelOp}_{@A,@B})) \cdots)))$$

Finally,

$$\left(\bowtie_{\alpha_1^{-1}} (\bowtie_{\alpha_2^{-1}[q_1]} (\bowtie_{\alpha_3^{-1}[q_2]} (\cdots \bowtie_{\alpha_n^{-1}[q_{n-1}]} (\bowtie_{\mathsf{self}[q_n]} (S^{-1})) \cdots)))\right)^{-1}$$

is the desired inequality relation above. Using this algorithm inductively, Theorem 4.14 follows. □

Applying the first two parts of the previous proof to NavXPath yields:

PROPOSITION 4.15 [GOTTLOB ET AL. 2005]. *A NavXPath NodeSet query Q can be evaluated on σ_{nav}-structures $\mathcal{D}$ in time $O(|\mathcal{D}| \cdot |Q|)$ and space $O(|\mathcal{D}|)$.*

Note that this improves the linear data complexity bound of Corollary 4.4.

Beyond FOXPath, we are faced with queries containing possibly nested numeric expressions involving the arithmetic operations $+$ and $*$ (whose graphs are infinite) and aggregations. For that reason, it is helpful to digress from the framework used above (i.e., relations $\subseteq A^2$ or $\subseteq A$) and view every expression e of *type* t (either *NodeSet*, Boolean, or Int) as defining a table $\{(n, [\![e]\!]_t(n)) \mid n \in A\}$. Each node n denotes a context in which expression e evaluates to value $[\![e]\!]_t(n)$. Thus such tables were called *context-value tables* in [Gottlob et al. 2005]. The context-value table of an expression e can be efficiently computed from the context-value table of the direct subexpressions of e. For FOXPath, the method for doing so was given in the previous proof, up to the notational subtleties that now for *NodeSet*-typed expressions, the value column may hold sets (nodes grouped by their context) while in the proof the relations defined were flat, and that context-value tables for Boolean-valued expressions are binary, with either "true" or "false" in the value column.

This method can be adapted to AggXPath without a runtime penalty, since on a binary relation $[\![p]\!]$ over the domain of nodes – and thus of quadratic size – the relations $\{(n,i) \mid [\![\text{count}(p)]\!]_{Int}(n) = i\}$ and $\{(n,i) \mid [\![\text{sum}(p/@A)]\!]_{Int}(n) = i\}$ can be computed in quadratic time without difficulty. For the arithmetic operation $*$ (multiplication), numbers can grow linearly with the query, thus a binary relation representing the result of a numeric relation may be of size $O(|A| \cdot |Q|)$. Thus,

PROPOSITION 4.16. *The AggXPath NodeSet queries Q can be evaluated on σ_{dom}-structures with domain A in time $O\big(|A| \cdot (|A|+|Q|) \cdot |Q|\big)$ and space $O\big(|A| \cdot (|A|+|Q|)\big)$.*

So far we have been moving only moderately beyond queries obtained from hypertree decompositions. However, XPath (and OrdXPath) supports position arithmetics which require more sophisticated contexts than AggXPath, where contexts are simply nodes. For OrdXPath, a single context node is not sufficient; for instance, the expression "position() = last()" relies on the position of a node within a set and the cardinality of that set as contexts (see (P2') in Section 2).

We extend context-value tables to be sets of tuples (n, j, k, v), where n is a context node, j and k are integers denoting a position j in and the size k of a set of nodes, v is a value, and the contexts n, i, k identify their tuples.

Values (including strings and numbers) were shown in [Gottlob et al. 2005] to remain small in XPath. The algorithm of [Gottlob et al. 2005] inductively computes context-value tables $\{(n, j, k, v) \mid [\![e]\!]_{Type(e)}(n, j, k) = v\}$ for each subexpression e of a query bottom-up. Taking into context all the built-in functions of XPath, this yields the following upper bound.

THEOREM 4.17 [GOTTLOB ET AL. 2005]. *Full XPath 1.0 is in time $O(|A|^5 \cdot |Q|^2)$.*

We state this result without a proof and refer to [Gottlob et al. 2005] for the formal definition of full XPath 1.0 and the proof, which are beyond our scope and yield little further insight. Improvements yielding somewhat better bounds can be found in [Gottlob et al. 2005].

EXAMPLE 4.18. Consider the numerical expression position() $* \, 2 <$ last(). We compute the context-value tables of its subexpressions bottom-up as

$$\begin{aligned}
CVT_{\text{position()}} &:= \{(n, j, k, j) \mid (n, j, k) \text{ a context}\} \\
CVT_{\text{position()}*2} &:= \{(n, j, k, 2 * v) \mid (n, j, k, v) \in CVT_{\text{position()}}\} \\
CVT_{\text{last()}} &:= \{(n, j, k, k) \mid (n, j, k) \text{ a context}\} \\
CVT_{\text{position()}*2<\text{last()}} &:= \{(n, j, k, (v_1 < v_2)) \mid (n, j, k, v_1) \in CVT_{\text{position()}*2}, \\
&\qquad (n, j, k, v_2) \in CVT_{\text{last()}}\}
\end{aligned}$$

In summary, there is a close connection between the context-value table-based dynamic programming algorithm of [Gottlob et al. 2005] and the hypertree-width based techniques presented before. However, beyond the difficulties dealt with in the proof of Theorem 4.14, XPath supports built-in functions (e.g. arithmetic and string functions) whose graphs are infinite, as well as aggregations, so non-trivial extensions of hypertree decomposition techniques are needed to obtain the PTime combined complexity of XPath.

We summarize the time complexity bounds in the following table; below the input is assumed to be a σ_{dom} structure $\mathcal{D}$ with domain A:

Fragment	Complexity
NavXPath	$\lvert\mathcal{D}\rvert \cdot \lvert Q\rvert$ (Proposition 4.15)
FOXPath	$\lvert A\rvert^2 \cdot \lvert Q\rvert$ (Theorem 4.14)
AggXPath	$\lvert A\rvert \cdot (\lvert A\rvert + \lvert Q\rvert) \cdot \lvert Q\rvert$ (Proposition 4.16)
XPath 1.0	$\lvert A\rvert^5 \cdot \lvert Q\rvert^2$ (Theorem 4.17)

4.5 Parallel Complexity

Now that the combined complexity of XPath is known to be polynomial, one may ask whether XPath is also PTime-hard, or alternatively, whether it is in the complexity class NC and thus effectively parallelizable. Apart from theoretical interest, a precise characterization of XPath evaluation in terms of parallel complexity classes may lead to a better understanding of what computational resources are necessarily required for query evaluation. For example, it is strongly conjectured that all algorithms for solving PTime-hard problems actually require a polynomial amount of working memory. However, performing XPath query evaluation with limited memory resources is important in practice, for instance in the context of data stream processing.

For an upper bound for conjunctive FOXPath, we can use the following result about conjunctive queries of bounded hypertree-width together with our Theorem 4.10.

Theorem 4.19 [Gottlob et al. 2001]. *The conjunctive queries of bounded hypertree-width over arbitrary relational structures are in* LogCFL *with respect to combined complexity.*

Corollary 4.20. *Conjunctive* FOXPath *is in* LogCFL *(combined complexity).*

In [Gottlob et al. 2005], LogCFL membership is proven for a much larger fragment of XPath without negation which even supports arithmetics and aggregations. Here we give a direct proof for positive FOXPath.

Proposition 4.21 [Gottlob et al. 2005]. *Positive* FOXPath *is in* LogCFL *with respect to combined complexity.*

Proof Idea. By an encoding as a NAuxPDA that runs in polynomial time using a LogSpace worktape. We will actually show how to use a NAuxPDA to compute the set of nodes to which an XPath query evaluates, even though the complexity class LogCFL is defined in terms of decision problems and for the above-mentioned lower bound only a decision problem (e.g. that of checking whether a given node is selected by an XPath query) makes sense.

We will use the symbol & for creating references and $*$ to dereference them. We will associate each query with its (binary) parse tree obtained in the usual fashion, using grammar rules $p ::= \mathit{axis} :: A[q]/p \mid \mathit{axis} :: A[q]$ to parse paths (i.e., producing a right-deep tree for a path). An example of such a parse tree is shown in Figure 5. We identify nodes of the query tree with the expressions their subtrees represent. For a path expression p, we use $sel(v_Q)$ to denote the rightmost leaf in the subtree of the query tree corresponding to p; thus $sel(v_Q)$ denotes the "right tip" of the path which selects nodes.

We use four log-space registers that will be kept on the worktape, sel (to iterate over the nodes of the data tree and check which are to be selected by the query), v_t (to hold a node from the data tree), r_{val} (for a pointer to a data value in the data tree, represented by an integer indicating the starting position of the data value's representation inside the representation of the data tree), and v_Q (for a current node from the parse tree of the query) on the worktape.

The evaluation of the query proceeds by iterating over all the nodes of the data tree (using register sel), and for each node does a single depth-first left-to right traversal of its parse tree, starting with v_Q the root node of the query tree, v_t the root of the input tree, and $r_{val} = \perp$.

By default, query tree nodes v_Q with two children are processed as follows. First we put (v_Q, v_t, r_{val}) onto the stack. Then we process the first child of v_Q. On returning we take (v_Q, v_t, r_{val}) off the stack (and set the registers). Finally process the second child of v_Q.

There are a few exceptions. When $v_Q = \alpha{::}A[q]/p$ and $v_t = n$, we first put n on the stack, nondeterministically guess a node n' such that $\alpha(n, n')$ and $A(n')$, set v_t to n', and only then we process the two children as just described. Expressions $p/@A/deref()$ are handled similarly.

For $p/@A = p'/@B$, r_{val} is not put on the stack before and taken off the stack after processing the first child. When arriving at $sel(p)$, we set r_{val} to $@A(v_t)$. When arriving at $sel(p')$, we verify that $r_{val} = @B(v_t)$.

If $v_Q = q \vee q'$, we nondeterministically choose either q or q' and verify that it holds relative to the current position v_t.

At $sel(p)$, where p is the query, we check whether $v_t = sel$. If so, we output node sel as a result.

It is not difficult to verify that this nondeterministic algorithm runs on an NAuxPDA in polynomial time, using only logarithmic space on the worktape. □

EXAMPLE 4.22. The FOXPath query .//A[.//B/@C = D[E/@F = G/@H]/@I] can be evaluated using a NAuxPDA given by the following pseudocode: (1) Guess w such that $[\![.//A]\!](v_t, w)$; $v_t := w$; (2) push v_t; (3) guess w such that $[\![.//B]\!](v_t, w)$; $v_t := w$; (4) $r_{val} := \& v_t.@C$; (5) $v_t :=$pop; (6) guess w such that $[\![./D]\!](v_t, w)$; $v_t := w$; push r_{val}; push v_t; (7) push v_t; (8) guess w such that $[\![./E]\!](v_t, w)$; $v_t := w$; (9) $r_{val} := \& v_t.@F$; (10) $v_t :=$pop; (11) guess w such that $[\![./G]\!](v_t, w)$; $v_t := w$; (12) check that $* r_{val} = v_t.@H$; (13) $v_t :=$pop; $r_{val} :=$pop; (14) check that $* r_{val} = v_t.@I$; (15) accept.

Note that this program is faithful to the construction mentioned above except that we do not push or pop the v_Q register (the query has been compiled into the program).

The fact that the run of this NAuxPDA is intuitively a depth-first traversal of the parse tree of the query is illustrated in Figure 5. □

It was shown in [Gottlob et al. 2005] by a reduction from the SAC^1 circuit value problem that the LOGCFL upper bound of Theorem 4.21 is tight: positive NavXPath is LOGCFL-complete with respect to combined complexity.

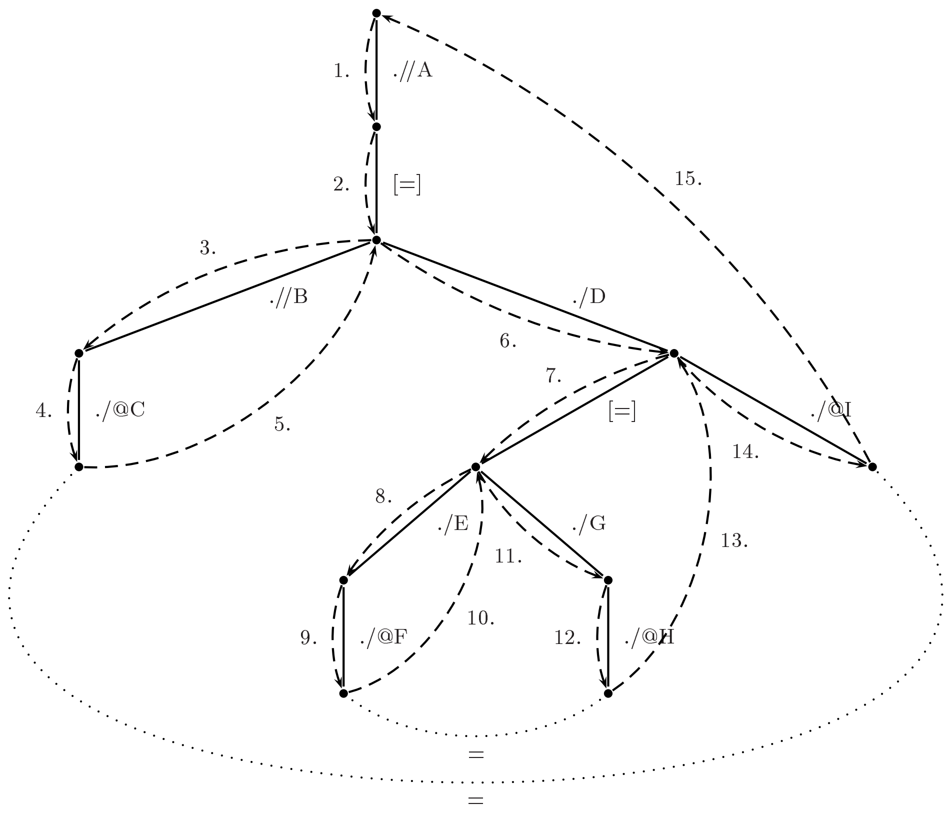


Fig. 5. NAuxPDA run for query .//a[.//b/@c = d[e/@f = g/@h]/@i].

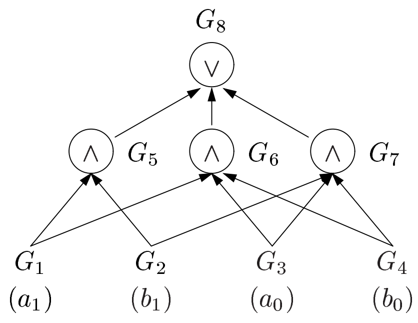


Fig. 6. A 2-bit full adder carry-bit circuit.

Unfortunately, the positive result on the parallel complexity of positive XPath does not extend to full XPath, or even NavXPath.

Theorem 4.23 [Gottlob et al. 2005]. *NavXPath is PTime-hard (combined complexity).*

Proof. The proof is by reduction from the *monotone Boolean circuit value* problem, which is PTime-complete. Note that the classical reduction from PTime-bounded Turing machines to (monotone) Boolean circuits proving this (see e.g. the proof of Theorem 8.1 in [Papadimitriou 1994]) only produces *layered* circuits.[4]

Given an instance of this problem, a monotone Boolean circuit and a mapping θ that assigns either 0 or 1 to each of the input gates, let M denote the number of

[4]A circuit is called layered is there is a mapping l that assigns to each gate an integer such that if there is an edge from gate G_i to G_j, then $l(G_j) = l(G_i) + 1$.

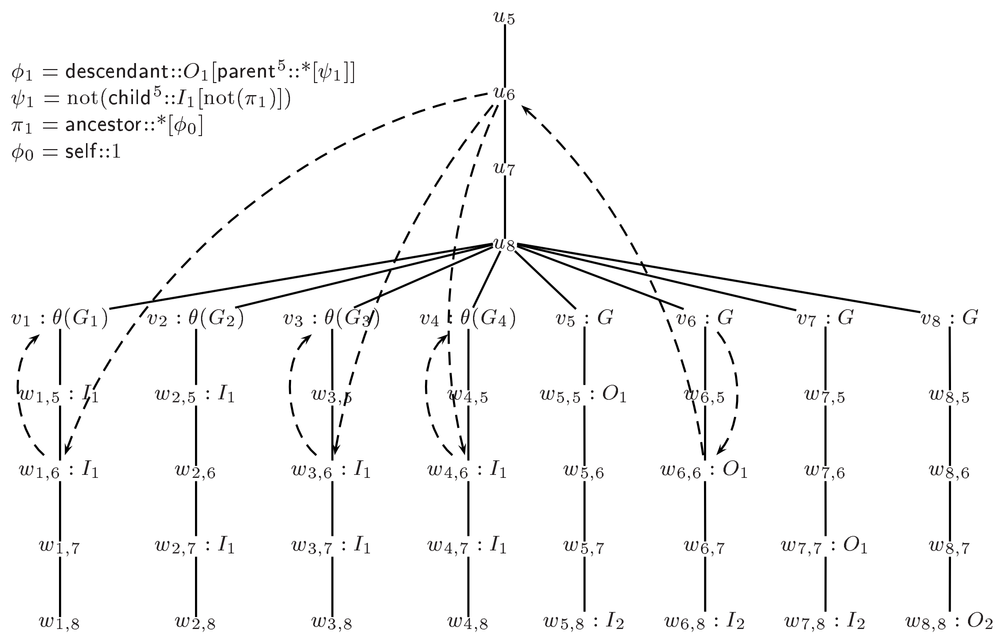


Fig. 7. Document tree corresponding to the carry-bit circuit. The figure also illustrates that $[\![\phi_1]\!]_{Boolean}(v_6) \Leftrightarrow \theta(G_1) = 1 \wedge \theta(G_3) = 1 \wedge \theta(G_4) = 1$.

input gates and let $N \geq 1$ denote the number of all other gates in the circuit (the *internal* gates). Let K be the number of layers in the circuit, that is, the height of the circuit. Let the gates be named $G_1 \dots G_{M+N}$. Without loss of generality[5], we may assume that the gates $G_1 \dots G_{M+N}$ are numbered in some order such that no gate G_i depends on the output of another gate G_j with $j > i$. In particular, the input gates are named $G_1 \dots G_M$ and the output gate is G_{M+N}. We may assume that there is precisely one gate at the topmost layer K, the output gate.

Figure 6 shows an example of a circuit with appropriately numbered gates. This circuit computes the carry-bit of a two-bit full-adder, that is, it tells whether adding the two-bit numbers a_1a_0 and b_1b_0 leads to an overflow. The carry-bit c_1 is computed as $(a_1 \wedge b_1) \vee (a_1 \wedge c_0) \vee (b_1 \wedge c_0)$ where $c_0 = a_0 \wedge b_0$ is the carry-bit of the lower digit (a_0 and b_0).

For a given instance of the monotone Boolean circuit value problem, we compute a pair consisting of a document tree and a NavXPath query as follows.

The **document tree** consists of nodes u_j, v_i, and $w_{i,j}$ for all $1 \leq i \leq M+N$, $M+1 \leq j \leq M+N$. The root node is u_{M+1}, and there are edges

—from u_j to u_{j+1} for $M+1 \leq j < M+N$,

—from u_{M+N} to v_i and from v_i to $w_{i,M+1}$ for all $1 \leq i \leq M+N$, and

—from $w_{i,j}$ to $w_{i,j+1}$ for all $1 \leq i \leq M+N$, $M+1 \leq j < M+N$.

Node labels are taken from the alphabet $\Sigma = \{0, 1, G, I_1, \dots, I_K, O_1, \dots, O_K\}$ and each tree node is assigned at most one such label. (We allow for "unlabeled" nodes, which can be considered to simply carry a label not from Σ.) This is done as follows. Each node out of v_i for $1 \leq i \leq M$ is assigned $\theta(G_i)$ as a label (either 0 or 1). The nodes $v_{M+1} \dots v_{M+N}$ are each assigned the label G. We assign label I_k to node $w_{i,j}$ iff internal gate G_j is in layer $1 \leq k \leq K$ and takes input from gate G_i. We assign label O_k to node $w_{j,j}$ iff internal gate G_j is in layer k. For our carry-bit example of Figure 6 with $M = 4$ and $N = 4$, the data tree is as shown in Figure 7,

[5]The gates can be "sorted" to adhere to such an ordering in logarithmic space. This is trivial if the circuit is layered, which we may assume by the observation made above.

where $\theta(G_1), \ldots, \theta(G_4) \in \{0, 1\}$ are the truth values a_1, b_1, a_0, and b_0, respectively, at the input gates.

In the following, we will abbreviate the n-times repeated application of an axis χ, $(\chi\text{::*}/)^{n-1}\chi\text{::*}$, as $\chi^n\text{::*}$. By χ^n::c, we denote $(\chi\text{::*}/)^{n-1}\chi$::c.

The **query** evaluating the circuit is

$$/\mathsf{descendant}::G[\phi_K]$$

with the condition expressions

$$\begin{aligned}
\phi_k &:= \mathsf{descendant}::O_k[\mathsf{parent}^{N+1}::*[\psi_k]] \\
\psi_k &:= \begin{cases} \mathsf{child}^{N+1}::I_k[\pi_k] & \ldots \text{ layer } k \text{ consists of } \vee\text{-gates} \\ \mathrm{not}(\mathsf{child}^{N+1}::I_k[\mathrm{not}(\pi_k)]) & \ldots \text{ layer } k \text{ consists of } \wedge\text{-gates} \end{cases} \\
\pi_k &:= \begin{cases} \mathsf{ancestor}::G[\phi_{k-1}] & \ldots\ k > 1 \\ \mathsf{ancestor}::*[\phi_{k-1}] & \ldots\ k = 1 \end{cases}
\end{aligned}$$

for $1 \leq k \leq K$ and $\phi_0 := \mathsf{self}::1$.

It uses the intuition of processing the circuit one layer at a time.

We will check whether our query on our document includes the particular node v_{M+N}. Indeed, by our construction, the query will select node v_{M+N} iff the circuit evaluates to true, and no other node will be selected.

It is easy to see that the reduction can be effected in LogSpace. We next argue that it is also correct.

The ϕ_k, ψ_k, and π_k are condition expressions (qualifiers), and we have already given a formal meaning $[\![\phi_k]\!]_{Boolean}(w)$ to the notion "ϕ_k matches node w" or equivalently "node w satisfies ϕ_k" (and analogously to $[\![\psi_k]\!]_{Boolean}(w)$ and $[\![\pi_k]\!]_{Boolean}(w)$).

Claim. *Let $0 \leq k \leq K$. Then, for all gates G_i in layer k,*

$$[\![\phi_k]\!]_{Boolean}(v_i) \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

This can be shown by an easy induction.

Induction start ($k = 0$). The gates of layer 0 are the input gates. By definition, an input gate G_i is true iff node v_i is labeled 1. but on precisely these nodes $\phi_0 = \text{self::1}$ is true. Thus our claim holds for $k = 0$.

Induction step. Now assume that our claim holds for ϕ_{k-1}. We show that it also holds for ϕ_k.

To start, it is easy to see that for all i, j,

$$[\![\pi_k]\!]_{Boolean}(w_{i,j}) \Leftrightarrow [\![\phi_{k-1}]\!]_{Boolean}(v_i).$$

Now observe that by our construction of the data tree, the nodes $w_{1,j}, \ldots, w_{j,j-1}$ encode the connections of gate G_j with its inputs. Gate G_i is an input to gate G_j if and only if node $w_{i,j}$ is labeled I_k, for k the layer of gate G_j. The node $w_{j,j}$ is labeled O_k. Observe also that the node u_j is precisely $N + 1$ levels above the nodes $w_{1,j}, \ldots, w_{M+N,j}$ in the data tree.

For $\vee$-gate G_j in layer k,

$$\begin{aligned}
[\![\psi_k]\!]_{Boolean}(u_j) &\Leftrightarrow \exists i\ I_k(w_{i,j}) \wedge [\![\pi_k]\!]_{Boolean}(w_{i,j}) \\
&\Leftrightarrow \text{gate } G_i \text{ is an input to } G_j \text{ and } G_i \text{ is true}
\end{aligned}$$

for $\wedge$-gate G_j in layer k,

$$\begin{aligned}
[\![\psi_k]\!]_{Boolean}(u_j) &\Leftrightarrow \forall i\ I_k(w_{i,j}) \rightarrow [\![\pi_k]\!]_{Boolean}(w_{i,j}) \\
&\Leftrightarrow \text{all inputs to } G_j \text{ are true}
\end{aligned}$$

Finally, since

$$[\![\phi_k]\!]_{Boolean}(v_j) \Leftrightarrow [\![\psi_k]\!]_{Boolean}(u_j),$$

our claim is shown for ϕ_k, $0 \leq k \leq K$.

Figure 7 illustrates the computation of the truth value of gate G_6 of our circuit example.

The overall query /descendant::G$[\phi_K]$ has a nonempty result (consisting of precisely the node v_{M+N}) exactly if the output gate G_{M+N} of the circuit evaluates to true, because G_{M+N} is the only gate in layer K, v_{M+N} is the only node labeled G that has an O_K descendant, and $[\![\phi_K]\!]_{Boolean}(v_{M+N})$ if and only if G_{M+N} evaluates to true.

In summary, we have provided a LogSpace reduction that maps any monotone Boolean circuit to a NavXPath query and a document tree such that the query evaluated on the tree returns node v_{M+N} precisely if the circuit evaluates to true. As the monotone Boolean circuit value problem is PTime-complete, our theorem is proven. □

Note that the above proof of the PTime lower bound does not employ axis steps with multiple qualifier brackets $axis[\cdot]\ldots[\cdot]$; indeed, as observed before, even for AggXPath, $axis[q_1]\ldots[q_n]$ is equivalent to $axis[q_1 \wedge \cdots \wedge q_n]$, but this is not true for OrdXPath. And indeed, the interaction of multiple qualifier brackets and position arithmetics has an impact on the complexity of XPath:

Theorem 4.24 [Gottlob et al. 2005]. *Positive* OrdXPath *is* PTime*-hard with respect to combined complexity.*

The PTime-hardness result actually only uses a fragment of OrdXPath with last() and steps with multiple qualifier brackets, but without position() or aggregation operations.

We give a brief overview over the remaining complexity results known for XPath. First, the PTime-hardness result of Theorem 4.23 essentially depends on the presence of both single-step axes and transitive axes: NavXPath using only the child and parent axes is in LogSpace with respect to combined complexity [Gottlob et al. 2005]. Tree patterns (conjunctive NavXPath) using only the descendant axis are in LogSpace as well [Götz et al. 2007].

The data complexity of XPath depends on encodings. XPath 1.0 on DOM trees (pointer structures) is LogSpace-complete if the concatenation operation on strings and multiplication are excluded from the language.

So far, we have always assumed that the input is basically given as a pointer structure (using signature σ_{dom}). But XML documents can also be considered in their natural textual (string) representation. The distinction is only relevant for the very small complexity class inside LogSpace, for which completeness is usually defined in terms of reductions not strong enough to map between DOM trees and strings. On string representations, NavXPath was shown to be in TC^0 [Gottlob et al. 2005], a complexity class inside LogSpace. Of course, on a relational encoding of the tree with all binary axis relations part of the encoding, FOXPath is first-order and inherits its AC^0 upper bound (yet inside TC^0) on the data complexity.

The query complexity of XPath 1.0 is in LogSpace [Gottlob et al. 2005]. This is a slightly curious fact. While for virtually all known traditional query languages, the query complexity is greater than the data complexity by at least an exponential factor (cf. e.g. [Abiteboul et al. 1995]), this is not the case of XPath.

4.6 Stream Processing

Because of the role of XML as a data exchange format, the problem of evaluating XPath on streaming XML data has attracted quite some research work.

A *streaming algorithm* scans its input data once – and only once – from left to right. Since data streams for practical purposes can be assumed to be infinitely long, one usually assumes that main memory is a limited resource. We can formalize streaming computation using a deterministic Turing machine with

—a read-only input tape on which the read head cannot move to the left,
—a write-only output tape on which the write head cannot move to the left, and
—a read/write work tape.

The resource of the greatest interest in this formal model is the space used on the work tape. Of course, the running time of the Turing machine is important as well. However, processing XPath is not an intrinsically hard problem: as explained in this work, it can be solved in main-memory in polynomial combined complexity, hence in particular in polynomial time in the data. The time upper bounds in terms of the data does not change when we move to the more restrictive streaming model. To our knowledge, no technique in the streaming XML literature requires running time greater than polynomial in the input (stream). Ideally, streaming algorithms should cope with a fixed amount of memory, independent of the input, but as we will see below, constant memory is not sufficient for evaluating even the simplest XPath queries.

To begin with we will focus our attention on the *XPath filtering problem*, for which better guarantees can be made. The filtering problem is the problem of testing whether a given XPath query relative to the root node has any matches (i.e., the problem of testing whether $[\![p]\!]_{Boolean}(root)$ is true for query p). The usual scenario is that of a stream of XML document*s* and a set of XPath queries describing subscriptions to documents on the stream matching the XPath queries, and has been referred to by *Selective Dissemination of Information*. This problem has been considered in [Altinel and Franklin 2000; Chan et al. 2000; Green et al. 2003; Diao et al. 2002] with the additional difficulty that algorithms have to scale to very large numbers – even millions – of queries to be matched in parallel.

Starting with [Bar-Yossef et al. 2007], techniques from communication complexity have been used for studying memory lower bounds of streaming XPath evaluation algorithms [Bar-Yossef et al. 2007; 2005; Grohe et al. 2007]. We only give one such lower bound result which uses the standard notion of complexity for XPath queries. We denote the depth of a tree $\mathcal{T}$ by $depth(\mathcal{T})$. It has been observed that

PROPOSITION 4.25 [GROHE ET AL. 2007]. *There can be no streaming algorithm with memory consumption $o(depth(\mathcal{T}))$, where $\mathcal{T}$ is the data tree, for the* CoreXPath *filtering problem.*

Of course, there are trees whose depth is linear in their size, so one can read this result in the sense that there can be no streaming algorithm for NavXPath that takes space less than linear in the *size* of the XML stream, so memory-efficient – and thus scalable – streaming XPath filtering is, from a certain point of view, in the worst case impossible.

Fortunately, XML trees tend to be shallow in practice, so showing this lower bound to be tight would be considered a positive result. As discussed early in this section, bottom-up tree automata allow to check MSO sentences in a single traversal of the tree. Using automata-based techniques, checking MSO queries in streaming fashion, and thus solving the XPath filtering problem, is feasible using only memory of size bounded by the depth of the tree (which in practice, for XML, is small).

THEOREM 4.26 (IMPLICIT IN [NEUMANN AND SEIDL 1998; SEGOUFIN AND VIANU 2002]). *Let $\mathbf{T}$ be a tree-language. If $\mathbf{T}$ is definable by an MSO-sentence over vocabulary σ_{nav}, then $\mathbf{T}$ can be recognized by a streaming algorithm using memory $O(depth(\mathcal{T}))$, where $\mathcal{T}$ is the data tree.*

COROLLARY 4.27. *There is a streaming algorithm for the* CoreXPath *filtering problem with memory consumption $O(depth(\mathcal{T}))$.*

Of course, it remains to ask whether these algorithms use memory that is small in the size of the XPath expression being filtered. Automata are a natural target of compilation for stream processing. They can be executed very efficiently on the stream, and for most forms of automata one can analyze the runtime memory usage easily.

Translating XPath queries into deterministic pushdown automata has been studied in several works [Green et al. 2003; Gupta and Suciu 2003] (and slightly less obviously in [Altinel and Franklin 2000; Chan et al. 2000; Diao et al. 2002]). Deterministic pushdown automata also give depth-bounded space usage. The blow-up required to compute such automata is exponential in the filter, and the sources of this exponentiality were explored in [Green et al. 2003]. In that work the automata are modularized by separation into two components. There is a deterministic finite automaton (DFA, defined on words, not on trees) for the path expression which runs on the path from the root node of the data tree to the current data tree node. There is also a pushdown automaton, independent of the path expression, that acts as a controller for the DFA, managing the stack and advancing the DFA every time a new node in the stream is encountered.

The first work to present a streaming algorithm for the XPath filtering problem that takes only memory linear in the depth of the tree and runs in time and space polynomial in the size of (the data and) the query was [Olteanu et al. 2003; Olteanu 2007]. They provide an algorithm that gives good bounds for any PNavXPath filter with only "forward" axes – i.e. child, next-sibling, descendant, following.

There, the exponential size of automata is avoided by not compiling automata for managing and recognizing the subexpressions of an XPath query into a single "flat" automaton. These automata are instead kept apart, as a *transducer network*. A similar transducer-network based approach to streaming XPath processing was developed in [Peng and Chawathe 2003]. A different algorithm for polynomial-time streaming XPath processing was presented in [Josifovski and Fontoura 2005].

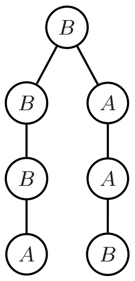
A transducer network consists of a set of synchronously running transducers (here, deterministic pushdown transducers, cf. [Hopcroft and Ullman 1979]) where each transducer runs, possibly in parallel with some other transducers, either on the input XML stream, or on the output of another transducer (in which case the input is the original stream where some nodes may have been annotated using labels). Two transducers may also be "joined", producing output whose annotations are pairs consisting of the annotations produced by the two input transducers.

We next formalize this and exhibit some of the transducers that form part of a transducer network.

XPath queries are first rewritten into nested filters with paths of length one; for instance, query child::A/descendant::B is first rewritten into child[lab() $= A \wedge$ descendant[lab() $= B$]]. To emphasize that we do not aim to compute nodes matched by a path but to check whether the query can be successfully matched, we will write axis filters as $\exists$child[ϕ] and $\exists$descendant[ϕ]. The rewritten queries will now be translated into transducer networks inductively.

A deterministic pushdown transducer T is a tuple $(\Sigma, \Gamma, \Omega, Q, q_0, F, \delta)$ with input alphabet Σ, stack alphabet Γ, output alphabet Ω, set of states Q, start state q_0, set of final states F, and transition function $\delta : Q \times \Sigma \times (\epsilon \cup \Gamma) \to Q \times \Gamma^* \times \Omega$. For determinism we require that for no $q \in Q, s \in \Sigma, \gamma \in \Gamma$, both $\delta(q, s, \epsilon)$ and $\delta(q, s, \gamma)$ are defined. Here ϵ denotes the empty word. All our transducers will have $Q = F$; that is, all states are final states, so all valid runs will be accepting. If the transducer T is in state q and has uv on the stack, and if $\delta(q, s, v) = (q', w, s')$, then T makes a transition to state q' and stack uw $(u, v, w \in \Gamma^*)$ on input s, and produces output o, denoted $(q, uv) \overset{s/o}{\to} (q', uw)$. A run on input $s_1 \dots s_n$ is a sequence of transitions $(q_0, \epsilon) \overset{s_1/o_1}{\to} \cdots \overset{s_n/o_n}{\to} (q, u)$ that produces output $o_1 \dots o_n$.

A transducer $T[\exists$descendant$[\phi]]$ running on the output stream of transducer $T[\phi]$ is a deterministic pushdown transducer with $\Sigma = \Omega = \{\langle\rangle, t, f\}$, $\Gamma = \{t, f\}$, $Q =$



$$\begin{tikzcd}
T[\phi_1 := (\mathsf{lab}() = B)] \arrow[d] & & \\
T[\phi_2 := \exists\mathsf{descendant}[\phi_1]] \arrow[dr] & & T[\phi_3 := (\mathsf{lab}() = A)] \arrow[dl] \\
& T[\langle \phi_2, \phi_3 \rangle] \arrow[d] & \\
& T[\phi_4 := \phi_2 \wedge \phi_3] &
\end{tikzcd}$$

	time $\longrightarrow$													
input stream	$\langle B\rangle$	$\langle B\rangle$	$\langle B\rangle$	$\langle A\rangle$	$\langle /A\rangle$	$\langle /B\rangle$	$\langle /B\rangle$	$\langle A\rangle$	$\langle A\rangle$	$\langle A\rangle$	$\langle /B\rangle$	$\langle /A\rangle$	$\langle /A\rangle$	$\langle /B\rangle$
transducer	synchronous output													
$T[\phi_1 := (\mathsf{lab}() = B)]$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	f	t	t	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	t	f	f	t
$T[\phi_2 := \exists\mathsf{descendant}[\phi_1]]$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	f	f	t	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	f	t	t	t
$T[\phi_3 := (\mathsf{lab}() = A)]$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	t	f	f	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	f	t	t	f
$T[\langle \phi_2, \phi_3 \rangle]$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	(f,t)	(f,f)	(t,f)	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	(f,f)	(t,t)	(t,t)	(t,f)
$T[\phi_4 := \phi_2 \wedge \phi_3]$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	f	f	f	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$	f	t	t	f

Fig. 8. Document tree (top left), transducer network (top right), and run of the transducer network (bottom).

$F = \{q_f, q_t\}$, $q_0 = q_f$, and transition function

$$\delta : \begin{cases} (q_x, \langle\rangle, \epsilon) \mapsto (q_f, x, \langle\rangle) \\ (q_x, y \in \{t, f\}, z) \mapsto (q_{x \vee y \vee z}, \epsilon, x). \end{cases}$$

On seeing an opening tag of a node, this transducer memorizes on the stack whether ϕ was matched in the subtrees of the previously seen siblings of that node. On returning (i.e., seeing a closing tag), the transducer labels the node (by its proxy the closing tag) with t or f (true or false) depending on whether ϕ was matched in the node's subtree, which is encoded in the state.

Example 4.28. On input $\langle\rangle\langle\rangle\langle\rangle\langle\rangle ftt\langle\rangle\langle\rangle\langle\rangle tfft$, $T[\exists\mathsf{descendant}[\cdot]]$ has the run

$$(q_f, \epsilon) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, f) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, ff) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, fff) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, ffff) \xrightarrow{f/f} (q_f, fff) \xrightarrow{t/f}$$
$$(q_t, ff) \xrightarrow{t/t} (q_t, f) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, ft) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, ftf) \xrightarrow{\langle\rangle/\langle\rangle} (q_f, ftff) \xrightarrow{t/f} (q_t, ftf) \xrightarrow{f/t}$$
$$(q_t, ft) \xrightarrow{f/t} (q_t, f) \xrightarrow{t/t} (q_t, \epsilon)$$

and produces output $\langle\rangle\langle\rangle\langle\rangle\langle\rangle fft\langle\rangle\langle\rangle\langle\rangle fttt$ (see Figure 8). □

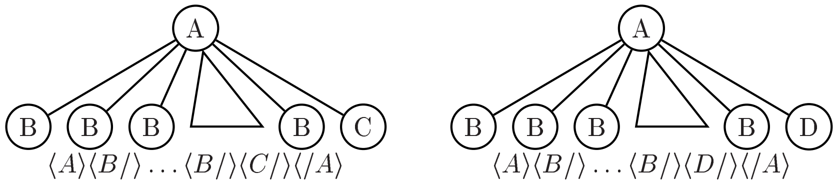
A transducer $T[\exists\mathsf{child}[\phi]]$ can be defined similarly.

The transducers for testing labels and computing conjunctions of filters do not need a stack. The transducer $T[\mathsf{lab}() = A]$ has the opening and closing tags of the XML document as input alphabet Σ, $\Omega = \{\langle\rangle, t, f\}$, $Q = F = \{q_0\}$, and $\delta = \{(q_0, \langle\cdot\rangle, \epsilon) \mapsto (q_0, \epsilon, \langle\rangle), (q_0, \langle /A\rangle, \epsilon) \mapsto (q_0, \epsilon, t), (q_0, \langle /B\rangle, \epsilon) \mapsto (q_0, \epsilon, f)\}$ (where B stands for all node labels other than A). The transducer $T[\phi \wedge \psi]$ has $\Sigma = \{\langle\rangle\} \cup \{t, f\}^2$, $\Omega = \{\langle\rangle, t, f\}$, $Q = F = \{q_0\}$ and $\delta = \{(q_0, \langle\rangle, \epsilon) \mapsto (q_0, \epsilon, \langle\rangle), (q_0, (x, y), \epsilon) \mapsto (q_0, \epsilon, x \wedge y)\}$.

The overall execution of a transducer network is exemplified in Figure 8, where the filter that matches the XPath expression self::A/descendant::B, rewritten into $(\exists\mathsf{descendant}[\mathsf{lab}() = B]) \wedge \mathsf{lab}() = A$ is evaluated using a transducer network. The transducers for the different subexpressions run synchronously; each symbol (opening or closing tag) from the input stream is first transformed by $T[\phi_1]$ and $T[\phi_3]$; the output of $T[\phi_1]$ is piped into $T[\phi_2]$ and the output of both $T[\phi_2]$ and

$T[\phi_3]$, as a pair of symbols, is piped into $T[\phi_4]$. Only then do we proceed to the next symbol of the input stream, which is handled in the same way, and so on. In the example of Figure 8, the final transducer labels exactly those nodes t on which the filter is true. Checking whether the filter can be matched on the root node, which is not the case in this example, can be done using an additional pushdown automaton – not exhibited here but simple to define.

We now comment on the problem of selecting nodes matched by XPath queries. We first note that any streaming algorithm will have to buffer most of the XML document in the worst case. Consider the following two trees.


A
B
B
B
B
C
$\langle A\rangle\langle B/\rangle\ldots\langle B/\rangle\langle C/\rangle\langle/A\rangle$
A
B
B
B
B
D
$\langle A\rangle\langle B/\rangle\ldots\langle B/\rangle\langle D/\rangle\langle/A\rangle$

Consider the query /child::A[child::C]/child::B. Any implementation of this query must select the B-nodes of the left tree but not those of the right tree. Hence such an implementation will have to buffer all B-children of the A-node before a C-node is seen (or not seen) on the stream. In the worst case this may amount to buffering almost all the nodes of the document.

The problem of selecting nodes using XPath on XML streams using polynomial time combined complexity and small space was studied in several works, including [Olteanu 2007; Peng and Chawathe 2003; Bar-Yossef et al. 2007; 2005; Ramanan 2005; Gou and Chirkova 2007]. The results in these papers are usually space bounds depending linearly on the depth of the data tree, a function of certain properties of the query (such as, e.g., *query frontier size* [Bar-Yossef et al. 2007]), and the number of candidate output nodes from the data tree: as we have seen immediately above, we can not hope to do better than this. The known bounds are for fragments of PCoreXPath with only forward axes.

4.7 Processing XPath in Databases

There has been much work on processing XPath (as a fragment on XQuery) and tree pattern queries on XML documents stored in *databases*, that is, in secondary storage, both in the context of native XML databases and even more so on relational representations of XML databases.

A topic related to XPath processing that has been addressed in many papers is storing XML data in a way that allows for efficient query processing and updates [Shanmugasundaram et al. 1999; Fiebig and Moerkotte 2000; Tatarinov et al. 2002; Grust et al. 2004; 2003; O'Neil et al. 2004; Weigel et al. 2005; May et al. 2006]. Clearly, once the data is to be stored in a database in a way other than a single monolithic document (i.e., text file) to allow for the addressing and indexing of data, the smaller data chunks (usually document tree nodes) require identifiers of some form. Much work has been done on finding appropriate schemes for storing XML data relationally (e.g. [Shanmugasundaram et al. 1999; Tatarinov et al. 2002]), but numbering schemes for XML nodes that assign unique identifiers to tree nodes that implicitly contain navigation information are also relevant in native XML database systems. It is implicit in [Tatarinov et al. 2002] that, when designing a node numbering scheme for XML data, a tradeoff is necessary between the scheme's support for efficient navigation (tree pattern queries) and the efficiency of processing updates. Numbering schemes in which the node identifiers contain much position information allow for more efficient query processing than do schemes which assign only local information that is relative to parent and ancestor nodes – but updates to the data are more likely to require a relabeling of many nodes with numbers.

Currently two numbering schemes have become prominent in most major research and commercial implementations. The first is the Dewey numbering scheme [Tatarinov et al. 2002; May et al. 2006] in which a node that is the j-th child of a node with identifier i is assigned the identifier $i.j$; thus the Dewey numbering scheme is the familiar scheme used to label hierarchies of sections and subsections in most books. Given a Dewey numbering scheme, the ancestors of a given node are completely determined and checking whether another node satisfies one of the axes is easily decided. The second [Fiebig and Moerkotte 2000; Grust et al. 2004; 2003] is a form of *global* numbering scheme (cf. [Tatarinov et al. 2002]). It assigns a preorder ($<_{\mathrm{pre}}$) and a postorder ($<_{\mathrm{post}}$) traversal index. In addition, the $<_{\mathrm{pre}}$-index of the parent is stored with each node. Here all axes can be computed using simple θ-joins. Thus the transitive axis relations, which would take space quadratic in the size of the tree if they had to be explicitly stored in the database, can be computed on demand using plain relational algebra, with no need for recursion.

As shown in Section 2.1, $<_{\mathrm{pre}}$ and $<_{\mathrm{post}}$ can be defined from $R_{\mathsf{descendant}}$ and $R_{\mathsf{following}}$. The converse is also possible:

$$\begin{aligned} R_{\mathsf{descendant}}(x, y) &:\Leftrightarrow x <_{\mathrm{pre}} y \wedge y <_{\mathrm{post}} x \\ R_{\mathsf{following}}(x, y) &:\Leftrightarrow x <_{\mathrm{pre}} y \wedge x <_{\mathrm{post}} y \end{aligned}$$

From these axis relations, all others can be defined in first-order logic. Thus, a node-labeled tree can be completely represented by one triple (i, j, a), consisting of a $<_{\mathrm{pre}}$-index i, a $<_{\mathrm{post}}$-index j, and a label a, for each node of the tree. (These indexes are chosen in a way that if two nodes u and v have, say, $<_{\mathrm{pre}}$-indexes i and i', then $i < i'$ iff $u <_{\mathrm{pre}} v$.)

This scheme does not require nodes to be labeled consecutively. Reasonable update performance can be achieved by not requiring $<_{\mathrm{pre}}$- and $<_{\mathrm{post}}$-indexes to be consecutive and initially leaving some indexes unused. Nodes can then be inserted by choosing a suitable pre- and postorder index from the unassigned indexes. A slight modification of this idea uses floating point numbers for the indexes; insertion is done by assigning $<_{\mathrm{pre}}$- and $<_{\mathrm{post}}$-numbers halfway between those of the nodes between which the new node is to be placed.

XML processing within databases focuses heavily on the case of conjunctive XPath and its extensions to XQuery. For queries on XML, one can distinguish between joins over data values and so-called *structural joins*. The latter are used to compute tuples of document nodes that are in a structural relationship to each other which can be described by a CoreXPath path expression, for instance pairs of nodes and their "A"-labeled descendents. While data value joins occur more frequently in XQuery, both kinds of joins can appear even in XPath. For example, the query of Example 4.9 contains four structural joins – corresponding to the four axis steps of the query – and one value join, which compares certain @D attribute values with @F attribute values. Many queries contain several structural joins that can be described by tree patterns (also called *twigs* in this context) and can be matched together.

As documented in the present section, pairs of nodes defined by CoreXPath expressions have special properties that give efficient structural join algorithms. The methods described in this survey have focused on a straightforward encoding of a tree as a relational structure. But efficient methods have also been discovered that either work for individual structural joins [Al-Khalifa et al. 2002; Grust et al. 2003] or holistically compute the matches of entire tree patterns [Bruno et al. 2002], for XML stored using the more sophisticated encodings discussed above. Note that in these encodings there is no need for a separate edge relation.

For XPath 1.0 the focus is on semi-joins. A key advantage of the twig query processing approach is that it extends the low complexity bounds of XPath to more general queries which return *all* query nodes in a match of a pattern, not just a single selected node. Such queries are important within the more general context of XQuery processing. The use of large-grained twig join operators and their integration into optimizers for XQuery is discussed in [Al-Khalifa and Jagadish 2002].

4.8 Further Bibliographic Remarks

The dynamic programming algorithm for full XPath 1 of [Gottlob et al. 2005] demonstrates in a rather straightforward way that XPath 1 can be evaluated in polynomial time. When introduced, this algorithm was the first of its kind, and it was observed that all XPath engines available at the time where taking exponential time in the worst case for evaluating XPath 1. However, the dynamic programming algorithm computes many useless intermediate results and consumes much memory. To fix this, a more efficient top-down algorithm is given in [Gottlob et al. 2005] as well. This algorithm still runs in polynomial time, with better worst-case upper bounds on running time and memory consumption. Further work on polynomial-time algorithms for full XPath 1 which elaborates on the results of [Gottlob et al. 2005] and integrates them into a native XML database management system can be found in [Brantner et al. 2005]. This work also shows how to integrate XQuery and efficient XPath processing using a single native algebra.

5. STATIC ANALYSIS

5.1 Satisfiability

Analysis of XPath originally focused on fragments of PNavXPath with only downward axes – basically, tree patterns (see Theorem 3.9). Such queries are always satisfiable, so analysis concentrated on the containment problem. However, as pointed out in [Benedikt et al. 2005], satisfiability becomes more difficult as soon as one has either negation or upward axes, or if one restricts trees to satisfy a schema, given for example, by a Document Type Definition (DTD). Simplifying for the purposes of this discussion, a DTD D can be thought of as a triple (Ele, P, r), where (1) Ele is a finite set of labels, ranged over by $A, B, \ldots$; (2) r is a distinguished label in Ele, called the *root type*; (3) P is a function that defines the labels of children for a given label A: for each A in Ele, $P(A)$ is a regular expression over Ele.

An XML-tree T *satisfies* (or *conforms to*) a DTD $D = (Ele, P, r)$, denoted by $T \models D$, if (1) the root of T is labeled with r; (2) each node n in T is labeled with a label in Ele, (3) for each node n of label $A \in ELE$, the list of labels of the children of n, listed from leftmost to rightmost, is in the regular language defined by $P(A)$.

To consider the impact of a DTD, fix n propositions $P_1 \ldots P_n$, and consider trees that are constrained to consist of 3 levels: a root element labeled with r, which has n children labeled $P_1 \ldots P_n$, with each P_i in turn having one child, which must be labeled with T or F. The DTD with root element r and productions $r \to P_1 \ldots P_n, P_1 \to T|F \ldots P_n \to T|F, T \to \epsilon, F \to \epsilon$ constrains a document to be of this form. Documents of this form code in an obvious way to valuations for the propositions $P_1 \ldots P_n$. If we take any CNF propositional formula $\phi = \bigwedge_i \bigvee_j \phi_{i,j}$ over $P_1 \ldots P_n$, we can write a corresponding negation-free CoreXPath qualifier that holds at the root of a tree iff the tree codes a model of ϕ. For example, $(P_1 \vee \neg P_2) \wedge (\neg P_1 \vee P_2)$ translates to $[(\mathsf{child}::P_1/\mathsf{child}::T \vee \mathsf{child}::P_2/\mathsf{child}::F) \wedge (\mathsf{child}::P_1/\mathsf{child}::F \vee \mathsf{child}::P_2/\mathsf{child}::T)]$. This argument shows:

PROPOSITION 5.1. *[Benedikt et al. 2005] It is NP-hard to check whether a* PNavXPath *expression with only the child axis is satisfiable with respect to a DTD.*

Satisfiability with respect to a DTD for PNavXPath turns out to be NP-complete: roughly speaking, one can guess a polynomial size satisfying tree using non-determinism and then verify that it is a satisfier by evaluating the XPath expression on it, which we know from the prior sections can be done in polynomial time. The line between tractability and intractability within PNavXPath is studied extensively in [Benedikt et al. 2005].

When general negation is added, as in NavXPath and CoreXPath, it is not immediately obvious that satisfiability is even decidable. One argument to establish decidability is via Proposition 3.1, and the fact that first-order logic over finite ordered labeled trees is known to be decidable [Thatcher and Wright 1968]. The standard proof of decidability for first-order logic is via an inductive translation into a tree automaton. Because complementation of an automaton requires an exponential blow-up in size at every negation step, the complexity of satisfiability for first-order logic over trees is known to be non-elementary [Thatcher and Wright 1968]. However, in the previous section we have shown that NavXPath Boolean queries translate into two-variable first-order logic. The satisfiability problem for FO^2 over arbitrary finite structures is known to be in NExpTime [Grädel et al. 1997]. In addition, [Grädel et al. 1997] shows that satisfiable FO^2 sentences have models of size exponential in the size of the sentence. However, this does not imply that the satisfiability problem for FO^2 is in NExpTime, since for this problem we have the constraint that the models must be trees (a constraint which is not expressible by an FO^2 sentence).

In [Etessami et al. 2002] it is shown that the satisfiability of FO^2 sentences over words is in NExpTime. We modify this below to show the satisfiability problem for trees is in NExpTime. Since the translation of NavXPath into FO^2 given in Section 3 is polynomial, we get a NExpTime bound for NavXPath.

Theorem 5.2. *There is an* NExpTime *algorithm deciding for a given sentence* $\phi \in FO^2$ *whether or not it is satisfiable by some ordered tree.*

Recall that Proposition 3.6 shows that unnested NavXPath$^\cap$, the extension of NavXPath with an intersection operator but where union may only occur on the top level, can be translated in polynomial time into FO^2. From this and Theorem 5.2, it follows that:

Corollary 5.3. *The satisfiability problem for unnested* NavXPath$^\cap$ *(and hence for unnested* NavXPath *and* CoreXPath*) is in* NExpTime.

We will see that this bound is not tight for NavXPath. We do not know the complexity of satisfiability for full NavXPath$^\cap$. A related language is PDL with an intersection operator, where the satisfiability problem has recently been shown to be 2-ExpTime hard even on one-letter trees [Lange and Lutz 2005]. However, this language is more expressive than NavXPath$^\cap$.

Since we know of no proof of Theorem 5.2 in the literature, we sketch one, following closely the approach of [Etessami et al. 2002]. First, we translate the problem of satisfiability on unranked trees to one on binary trees, using the standard encoding of an unranked tree as a binary tree. Let $FO^2[\sigma_{nav,bin}]$ be FO^2 over the unary signature Σ unioned with FChild, SChild (the first- and second-child relations of the binary tree representation), SChild*, $R_{\mathsf{descendant}}$. We consider a formula of $FO^2[\sigma_{nav,bin}]$ to be interpreted over *binary codes of unranked trees*, structures $T = (V, \ldots)$ in which *i)* (V, FChild $\cup$ SChild) is a tree of outdegree at most two, *ii)* each node is related to at most one node via FChild and at most one variable SChild, with these nodes being distinct, and *iii)* $R_{\mathsf{descendant}}$ is the transitive closure of FChild $\cup$ SChild, and SChild* is the transitive closure of SChild. The following is simple to show:

Proposition 5.4. *Satisfiability of* FO^2 *sentences over unranked trees is reducible in polynomial time to satisfiability of* $FO^2[\sigma_{nav,bin}]$ *sentences over binary codes of unranked trees.*

For an integer k, a *k-type* is a maximal consistent set of $FO^2[\sigma_{nav,bin}]$ formulas (in some fixed set of variables) where the maximal number of nested quantifiers (i.e. quantifier rank) is at most k. We will deal with k-types in 1 free variable, with such a type typically denoted $\tau(x)$. A binary code structure $(V, \ldots)$ is *k-compact* if:

—We do not have nodes $v_1, v_2 \in V$ with the same k-type, and with v_2 a descendant of v_1.

—Any two nodes with the same k-type have identical subtrees.

The next result shows that we can reduce satisfiability to a search for compact structures:

LEMMA 5.5. *An $FO^2[\sigma_{nav,bin}]$ sentence of quantifier rank $k > 1$ is satisfiable at the root of some binary code iff it is satisfiable at the root of a k-compact binary code.*

Proof. Let ϕ be an $FO^2[\sigma_{nav,bin}]$ sentence of quantifier rank k, and suppose ϕ is satisfiable in $B = (V, \ldots)$, and B is the structure of minimal size satisfying ϕ. Suppose there are nodes $v_1, v_2 \in V$ with the same k-type , with v_2 a descendant of v_1. Let S_1 be all nodes that are descendants of v_1 but are not descendants of v_2 (including v_2). Let B' be the code formed by removing all nodes in S_1 and attaching the subtrees of v_2 to v_1 (i.e. the first child of v_2 becomes the first child of v_1, etc.). Let f be the mapping from B' to B that maps a node beneath v_1 in B' to the corresponding node beneath v_2, and is the identity elsewhere on B'. We now show by induction on i that for each $i \leq k$, the i-type of a node $v \in B'$ is the same as the i-type of $f(v) \in B$.

For $i = 0$ this is clear, since the only atomic formulas in one variable are those that assert the label of a node, and the mapping f preserves labels. For the inductive step $i + 1$, note that a two-variable formula $\phi(x)$ of rank $i + 1$ can be taken to assert the existence or non-existence of a y with a certain axis relation to x and with a fixed i-type. All formulas asserting the non-existence of such a y are clearly preserved from x to $f(x)$, by induction. Suppose that for $x \in B'$ there is a y in B with i-type τ and with a given axis relationship to $f(x)$. If $y = f(w)$ for some w in B', then we can choose w as a witness to τ in B', since w will satisfy the same axis relation to x as y does to $f(x)$ (by definition of f), and will satisfy the same i-type as y by induction. Otherwise, it must be that y lies below v_1 but is incomparable to v_2. Since y lies below v_1 and v_2 has the same k-type in B (hence the same $i+1$-type) as v_1, there is y' below v_1 satisfying the same axes with respect to v_1 as y has to v_2, and such that the i-type of y' in B is the same as the i-type of y in B. Since y' is below v_1, $y' = f(w)$ for some $w \in B'$, and now we are done by induction.

The result of the construction above is a smaller tree in which the k-type of the root has the same type as in the original tree, thus violating minimality.

To get the second part of compactness, let Γ be the set of k-types $\tau(x)$ such that the second part is violated in B': that is, there are two nodes with type τ with distinct subtrees. We proceed by downward induction on $n = |\Gamma|$. If $n > 0$, choose a node $v \in B'$ satisfying a type in Γ that has maximal depth in the tree. Let τ be the k-type of v and S_v be the forest consisting of all descendants of v in B'. All nodes in S_v must satisfy a type outside of Γ. For every other node v' in B' satisfying τ, we replace the forest below v' with S_v (making the subtree below the first child of v into the subtree below the first child of v', etc.). Notice that the first condition of compactness (already holding of B') ensures that v' is not comparable to v. One can confirm by induction that the k-type of the root is unchanged by this substitution, by an argument identical to that used in the first part of this lemma. In this process, n is decreased by one, and hence the process terminates with a k-compact tree. □

From Lemma 5.5, Theorem 5.2 follows. The depth of a k-compact tree is at most the number of k-types, which is bounded by an exponential in ϕ. Furthermore, a k-compact tree can be represented via a DAG whose nodes are the k-types realized in the tree. Such a DAG represents the tree formed by duplicating shared subtrees. It is easy to see that one can check whether a given sentence is satisfied on a DAG representation of a tree in polynomial time. Our NExpTime algorithm just guesses a DAG structure on the k-types, and then confirms that the corresponding tree satisfies the sentence ϕ.

It is known that FO^2 is NExpTime-hard [Etessami et al. 2002]. The example showing NExpTime hardness from [Etessami et al. 2002] can be coded easily in unnested $\mathsf{NavXPath}^{\cap}$, hence we have that:

Theorem 5.6. *The satisfiability problem for unnested* $\mathsf{NavXPath}^{\cap}$ *is complete for* NExpTime.

From this proof, we get further information:

Corollary 5.7 to the proof of Theorem 5.2. *Let ϕ be an FO^2 sentence. If ϕ is satisfiable in some finite tree, then it is satisfiable in some tree of depth exponential in $|\phi|$ and size doubly exponential in $|\phi|$. The same holds for E an expression in unnested* NavXPath *extended with the intersection operator.*

Is this NExpTime-bound tight for NavXPath or CoreXPath? First note that the fact that FO^2 is NExpTime-hard does not imply the same for NavXPath, since the translation from FO^2 to NavXPath is exponential. [Marx 2004b] shows that satisfiability of NavXPath expressions can be decided in deterministic exponential time.

Theorem 5.8 [Marx 2004b]. NavXPath *satisfiability is decidable in* ExpTime. *Furthermore, since equivalence for* NavXPath *expressions can be reduced to satisfiability of a single expression, the equivalence problem can be decided in* ExpTime. *Since* CoreXPath *expressions can be mapped into* NavXPath *in linear time, these results hold for* CoreXPath *as well.*

[Marx 2004b] actually shows this for an extension of NavXPath that allows regular expressions on axes. Since the treatment in Marx's papers [Marx 2004b; 2004a; Afanasiev et al. 2005] is quite detailed, we give here only some comments on the proof. The proof is by reduction to the satisfiability problem for Deterministic Propositional Dynamic Logic (PDL) with Converse. PDL is similar to XPath, in that it is a modal language that allows the definition of binary relations (in dynamic logic "programs") as well as unary relations ("formulas"). As with XPath, the grammars for binary relations and unary relations are mutually recursive. Dynamic logics have a different data model than XPath, being defined over node and edge-labeled graphs. However, since formulas in the language can see only a part of the graph at a time, the behavior of the logic on general structures is closely related to its behavior on trees. Deterministic PDL with converse is formed over a set of atomic programs (analogous to axes in XPath) each of which is a function maps nodes in a graph to at most one other node. For each atomic program there is a "converse program" representing the inverse of the binary relation. In a binary tree the "first child" and "second child" relations are functional; hene we can interpret Deterministic PDL with Converse with two atomic program over binary trees, with the two programs chosen to be first and second child. Using the standard encoding of ordered unranked trees as binary trees, deterministic PDL with Converse over two programs can be interpreted on ordered trees. Because PDL allows new binary relations to be built up from old using regular expressions, the recursive axes, and in fact all of NavXPath (and more [Marx 2004b]), can be defined within it. Hence the satisfiability of XPath is reduced to the satisfiability problem fo Deterministic PDL with Converse sentences over binary trees. In [Vardi and Wolper 1986] it is shown that deterministic PDL with converse is decidable over all structures is in ExpTime. The proof relies on translating PDL programs into alternating automata on trees. [Marx 2004b] shows that the proof in [Vardi and Wolper 1986] can be modified to give the same bound over the class of codings of finite ordered trees. In [Afanasiev et al. 2005], a variant of PDL defined directly on ordered trees is given, which yields an alternate route (also going through [Vardi and Wolper 1986]) to the ExpTime bound.

[Neven and Schwentick 2003] shows that containment of NavXPath expressions is ExpTime-hard. An inspection of the proof shows that only CoreXPath expressions are needed for the hardness proof. Since containment of two (unnested) NavXPath expressions can be reduced to satisfiability of a single (unnested) expression, it follows that unnested NavXPath satisfiability is ExpTime-hard. Hence we see that the ExpTime bound is tight:

Corollary 5.9 combining [Neven and Schwentick 2003] and [Marx 2004b]. *The satisfiability problems for* CoreXPath*,* NavXPath*, and unnested* NavXPath *are all* ExpTime*-complete.*

5.2 Satisfiability for other XPath fragments

Now that we know that NavXPath and CoreXPath have ExpTime satisfiability, we can look at what happens as features are added or subtracted.

Better bounds can be obtained for sublanguages of NavXPath: Satisfiability of NavXPath with only child and parent is shown to be PSpace-complete in [Benedikt et al. 2005]. Satisfiability for PNavXPath is easily seen to be in NP (see [Hidders 2003]), and this is extended to PFOXPath in [Benedikt et al. 2005]. It is also shown in [Benedikt et al. 2005] that very simple fragments of PNavXPath have an NP-complete satisfiability problem – in the presence of both downward and upward axes, the problem is NP-complete, as well as in the presence of both left and right sibling axes. For PNavXPath with only downward axes, all expressions are clearly satisfiable; however, the satisfiability problem with respect to a given DTD can be NP-hard [Benedikt et al. 2005].

We now consider satisfiability as we move up in expressiveness from NavXPath. It is shown in [Benedikt et al. 2005] that the satisfiability of a FOXPath expression with respect to a DTD is undecidable. By using sibling axes instead of a DTD, one can see the following:

Theorem 5.10 [Geerts and Fan 2005]. *The satisfiability problem for* FOXPath *is undecidable.*

The proof uses a reduction from the halting problem for two-register machines which is known to be undecidable (see, e.g., [Börger et al. 1997]). Although full FOXPath is undecidable, the exact borderline of decidability is not well understood.

Question 5.11. *Is* FOXPath *without the sibling axes decidable?*

In fact, decidability is open even in the case of FOXPath with only child and parent.

One can also look at decidability on restricted classes of documents:

Question 5.12. *Is* FOXPath *decidable on documents with no branching (i.e. those where every element has at most one child)?*

5.3 Containment

The *containment problem* takes as input XPath expressions E and E', asking whether the output of E is contained in the output of E' on any source document at any node. Variations of the problem are *containment with respect to a DTD*, which takes a DTD as an additional argument, asking whether the above holds for E and E' over any source document satisfying the DTD. A special case of this is the *containment problem for a finite alphabet*, which takes a label alphabet Σ as additional parameter, asking whether containment holds for all source documents with labels in Σ.

The containment problem has been investigated extensively in the relational case for conjunctive queries, where it has close connections both to issues in data integration and query optimization, as well as to constraint satisfaction [Kolaitis and Vardi 2000; Gottlob et al. 2001]. The general conjunctive query containment problem is known to be NP-complete; however, many special cases are known to be in PTIME, including those in which the dependency graphs of the queries have bounded tree-width [Chekuri and Rajaraman 1997] or the queries have bounded hypertree-width [Gottlob et al. 1999]. In the case of conjunctive queries, containment of Q_1 in Q_2 reduces to determining whether Q_1 is satisfiable on an instance formed from Q_2, hence the complexity of containment is bounded by the combined complexity of evaluation. In the XPath setting there is no obvious correspondence between a query and a "canonical instance", and indeed the complexity of containment and evaluation turn out to be quite different.

Starting with the relational case as motivation, [Amer-Yahia et al. 2001; Miklau and Suciu 2002; Wood 2001] initiated the study of containment for XPath, beginning with subclasses of NavXPath without either the union operator or disjunction within filters (conjunctive NavXPath). The survey article of Schwentick [Schwentick 2004] gives a overview of the techniques used in getting bounds on containment; here we summarize only some of the results and the open questions. A modification of the minimal model technique for conjunctive queries shows that the containment problem for conjunctive Navigational XPath is in CO-NP – given queries P and Q one can generate a finite set of instances $I_i : i < n$ of size polynomial in P such that $P \subseteq Q$ iff each I_i satisfies Q [Miklau and Suciu 2002]. Since satisfaction can be checked in linear time, a CO-NP algorithm is simply to guess an I_i that fails to satisfy Q. In [Amer-Yahia et al. 2001], it is shown that for conjunctive NavXPath with only descendant axes the containment problem is in PTIME, while in [Wood 2001] it is noted that the same holds for conjunctive NavXPath with only child axes (indeed this last observation follows directly from the PTIME bounds for acyclic conjunctive queries in [Chekuri and Rajaraman 1997]). When both descendant axes and child axes are present the problem was shown to be CO-NP-complete [Miklau and Suciu 2002]. [Neven and Schwentick 2003] shows that the containment problem for conjunctive NavXPath with a finite alphabet is PSPACE-complete, while the containment problem with respect to a DTD is EXPTIME-complete. A finer analysis of the complexity of containment for conjunctive NavXPath with respect to a DTD and with respect to integrity constraints is given in [Wood 2003].

The complexity of containment for fragments of XPath larger than conjunctive NavXPath was studied by Neven and Schwentick. For PNavXPath, the general containment problem remains in CO-NP, while if the alphabet is fixed the problem is again PSPACE-complete [Neven and Schwentick 2003]. For full NavXPath, the containment problem, even with respect to a DTD, is in EXPTIME, since it is reducible to the satisfaction problem: this is noted in [Marx 2004b]. On the other hand, since [Neven and Schwentick 2003] shows that containment of NavXPath expressions is EXPTIME-hard, we have:

THEOREM 5.13 COMBINING [NEVEN AND SCHWENTICK 2003] AND [MARX 2004B]. *The containment problem for* NavXPath *is* EXPTIME-*complete, as is the containment problem for finite alphabet and the containment problem with respect to a DTD.*

When we turn to the XPath fragments with data values, the complexity of containment is not completely understood. The results of Deutsch and Tannen [Deutsch and Tannen 2001] imply that containment for PFOXPath is co-NP-complete, provided that the transitive sibling axes are not permitted and ”wildcard steps” (child steps with no restriction on the label) are disallowed. Their technique also yields a Π_2^P bound for full PFOXPath, although neither their terminology nor their fragments match PFOXPath exactly. They also establish Π_2^P bounds in the presence of integrity constraints called SXICs: these are incomparable to both finite alphabets and DTDs. [Deutsch and Tannen 2001] also provides lower bounds for containment in the presence of integrity constraints. Neven and Schwentick [Neven and Schwentick 2003] show that PFOXPath without sibling axes and without wildcard is in Π_2^P, and that the containment problem for PFOXPath extended with inequality is undecidable.

To our knowledge, the decidability of containment for general conjunctive FOXPath queries with respect to a DTD or a finite alphabet is open. Indeed we do not know whether one can decide containment of conjunctive queries over signature σ'_{dom} [6] in the presence of DTDs. The undecidability techniques of [Neven and Schwentick 2003] rely on disjunction, while [Deutsch and Tannen 2001] provides undecidability results with respect to integrity constraints. The upper bounds of both [Neven and Schwentick 2003; Deutsch and Tannen 2001] rely on the use of an infinite alphabet.

5.4 Further Bibliographic Remarks

While above we have dealt with the satisfiability and containment problems, a broader goal would be an algebraic simplification framework for XPath. [Benedikt et al. 2003] presents algebraic equations for simplification of XPath expressions. A system of equations is presented that is complete for equivalence of XPath expressions for a very small fragment (without filters and with only child axes). [Olteanu et al. 2002] gives a rewriting system geared not toward general equivalence, but for removing backward axes. [Amer-Yahia et al. 2001] deals not with equivalence but with optimization; it presents an algorithm for minimization of tree patterns in the presence of integrity constraints.

A natural question not addressed above is the implementation of satisfiability and containment tests for XPath. [Benedikt et al. 2005] implements a satisfiability test for a fragment of PNavXPath,in the presence of DTDs, based on a conversion to tree automata. [Lakshmanan et al. 2004] implements a satisfiability test for a tree pattern language that includes data value manipulation (incomparable in expressiveness with the XPath languages we consider here).

An additional static analysis problem is recognizing whether a query is in a given XPath fragment. In the context of navigational XPath, the problem of recognizing whether a first-order logic query is in NavXPath is open. This is closely-related to the (likewise open) problem of determining whether a tree automaton is equivalent to an FO^2 sentence . The problem of determining whether a first-order query over σ'_{dom} is in FOXPath is undecidable – this follows from the results of [Benedikt et al. 2005]. The problem of determining whether a conjunctive query over σ'_{dom} is expressible in conjunctive FOXPath has not been investigated (to our knowledge). Likewise, nothing is known concerning the problem of determining whether a first-order query (or a NavXPath query) is equivalent to a query in PNavXPath.

Acknowledgements: We thank Maarten Marx and Frank Neven for comments on this draft.

REFERENCES

Abiteboul, S., Hull, R., and Vianu, V. 1995. *Foundations of Databases.* Addison-Wesley.

[6]Recall that this is the relational signature with binary predicates for the graph of each attribute function, unary predicates for the labels, and binary predicates for the major axes.

Afanasiev, L., Blackburn, P., Dimitriou, I., Gaiffe, B., Goris, E., Marx, M., and de Rijke, M. 2005. "PDL for Ordered Trees". *Journal of Applied Non-Classical Logics 15*, 115–135.

Afanasiev, L., Franceschet, M., Marx, M., and de Rijke, M. 2004. "CTL Model Checking for Processing Simple XPath Queries". In *Proc. TIME.* 117–124.

Al-Khalifa, S. and Jagadish, H. V. 2002. "Multi-level operator combination in XML query processing". In *Proc. CIKM.* 134–141.

Al-Khalifa, S., Jagadish, H. V., Patel, J. M., Wu, Y., Koudas, N., and Srivastava, D. 2002. "Structural Joins: A Primitive for Efficient XML Query Pattern Matching". In *18th International Conference on Data Engineering (ICDE'02).*

Altinel, M. and Franklin, M. 2000. "Efficient Filtering of XML Documents for Selective Dissemination of Information". In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'2000).* Cairo, Egypt, 53–64.

Amer-Yahia, S., Cho, S., Lakshmanan, L. V., and Srivastava, D. 2001. "Minimization of Tree Pattern Queries". In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'01).* Santa Barbara, California, USA, 497–508.

Bar-Yossef, Z., Fontoura, M., and Josifovski, V. 2005. "Buffering in Query Evaluation over XML Streams". In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'05).*

Bar-Yossef, Z., Fontoura, M., and Josifovski, V. 2007. "On the Memory Requirements of XPath Evaluation over XML Streams". *Journal of Computer and System Sciences* **73,** 3, 391–441.

Barcelo, P. and Libkin, L. 2005. "Temporal logics over unranked trees". In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS).* 31–40.

Beauquier, D. and Pin, J.-E. 1989. "Factors of Words". In *Proc. ICALP.* 63–79.

Benedikt, M., Bonifati, A., Flesca, S., and Vyas, A. 2005. "Verification of Tree Updates for Optimization". In *Proceedings of the 17th International Conference on Computer Aided Verification.*

Benedikt, M., Fan, W., and Geerts, F. 2005. "XPath Satisfiability in the presence of DTDs". In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'05).*

Benedikt, M., Fan, W., and Kuper, G. 2003. "Structural Properties of XPath Fragments". In *Proc. of the 9th International Conference on Database Theory (ICDT).* Siena, Italy, 79–95.

Bird, S., Chen, Y., Davidson, S., Lee, H., and Zheng, Y. 2005. "Extending XPath to Support Linguistic Queries". In *PLAN-X.*

Börger, E., Grädel, E., and Gurevich, Y. 1997. *The Classical Decision Problem.* Springer.

Brantner, M., Helmer, S., Kanne, C.-C., and Moerkotte, G. 2005. "Full-fledged Algebraic XPath Processing in Natix". In *Proceedings of the 21st IEEE International Conference on Data Engineering (ICDE).*

Brüggemann-Klein, A., Murata, M., and Wood, D. 2001. "Regular Tree and Regular Hedge Languages over Non-ranked Alphabets: Version 1, April 3, 2001". Tech. Rep. HKUST-TCSC-2001-05, Hong Kong University of Science and Technology, Hong Kong SAR, China.

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

Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. 1990. "Symbolic Model Checking: 10^{20} States and Beyond". In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science (LICS).*

Carme, J., Niehren, J., and Tommasi, M. 2004. "Querying Unranked Trees with Stepwise Tree Automata". In *Rewriting Techniques and Applications.*

Chan, C. Y., Felber, P., Garofalakis, M. N., and Rastogi, R. 2000. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE).* San Jose, California, USA, February 26-March 1, 2002.

Chekuri, C. and Rajaraman, A. 1997. Conjunctive Query Containment Revisited". In *Proc. of the 6th International Conference on Database Theory (ICDT).* Delphi, Greece, 56–70.

Clarke, E. M., Grumberg, O., and Peled, D. 2000. *Model Checking.* MIT Press.

Courcelle, B. 1990. "Graph Rewriting: An Algebraic and Logic Approach". In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. 2. Elsevier Science Publishers B.V., Chapter 5, 193–242.

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

Diao et al., Y. 2002. "YFilter: Efficient and Scalable Filtering of XML Documents.". In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE).*

Doner, J. 1970. "Tree Acceptors and some of their Applications". *Journal of Computer and System Sciences* **4**, 406–451.

Etessami, K., Vardi, M., and Wilke, T. 2002. "First Order Logic with Two Variables and Unary Temporal Logic". *Information and Computation 179*.

Etessami, K. and Wilke, T. 2000. "An Until Hierarchy and Other Applications of an Ehrenfeucht-Fraisse Game for Temporal Logic". *Information and Computation 160*, 88–108.

Fan, W., Chan, C., and Garofalakis, M. 2004. Secure XML querying with security views. In *SIGMOD*.

Fiebig, T. and Moerkotte, G. 2000. "Evaluating Queries on Structure with eXtended Access Support Relations". In *Proc. WebDB*.

Flum, J., Frick, M., and Grohe, M. 2002. "Query Evaluation via Tree-Decompositions". *Journal of the ACM* **49***,* 6, 716–752.

Frick, M., Grohe, M., and Koch, C. 2003. "Query Evaluation on Compressed Trees". In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS).* Ottawa, Canada.

Geerts, F. and Fan, W. 2005. "XPath Satisfiability with Sibling Axes". In *Proc. 10th DBPL*.

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

Gottlob, G. and Koch, C. 2004. "Monadic Datalog and the Expressive Power of Web Information Extraction Languages". *Journal of the ACM* **51***,* 1, 74–113.

Gottlob, G., Koch, C., and Pichler, R. 2002. "Efficient Algorithms for Processing XPath Queries". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB).* Hong Kong, China, 95–106.

Gottlob, G., Koch, C., and Pichler, R. 2005. "Efficient Algorithms for Processing XPath Queries". *ACM Transactions on Database Systems* **30***,* 2 (June), 444–491.

Gottlob, G., Koch, C., Pichler, R., and Segoufin, L. 2005. "The Complexity of XPath Query Evaluation and XML Typing". *Journal of the ACM* **52***,* 2 (Mar.), 284–335.

Gottlob, G., Koch, C., and Schulz, K. U. 2004. "Conjunctive Queries over Trees". In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'04).* Paris, France, 189–200.

Gottlob, G., Leone, N., and Scarcello, F. 1999. "Hypertree Decompositions and Tractable Queries". In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'99).* 21–32.

Gottlob, G., Leone, N., and Scarcello, F. 2001. "The Complexity of Acyclic Conjunctive Queries". *Journal of the ACM* **48***,* 1, 431–498.

Gottlob, G., Leone, N., and Scarcello, F. 2002. "Hypertree Decompositions and Tractable Queries". *Journal of Computer and System Sciences* **64***,* 3, 579–627.

Götz, M., Koch, C., and Martens, W. 2007. "Efficient Algorithms for the Tree Homeomorphism Problem". In *Proc. 11th International Symposium on Database Programming Languages (DBPL).* Vienna, Austria.

Gou, G. and Chirkova, R. 2007. "Efficient algorithms for evaluating XPath over streams". In *Proc. SIGMOD*. 269–280.

Grädel, E., Kolaitis, P., and Vardi, M. 1997. "On the Decision Problem for Two-variable First-order Logic". *Bulletin of Symbolic Logic 3*, 53–69.

Green, T. J., Miklau, G., Onizuka, M., and Suciu, D. 2003. "Processing XML Streams with Deterministic Automata". In *Proc. of the 9th International Conference on Database Theory (ICDT).*

Greenlaw, R., Hoover, H. J., and Ruzzo, W. L. 1995. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press.

Grohe, M., Koch, C., and Schweikardt, N. 2007. "Tight Lower Bounds for Query Processing on Streaming and External Memory Data". *Theor. Comput. Sci.* **380***,* 1–2, 199–217.

Grust, T., van Keulen, M., and Teubner, J. 2003. "Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps". In *Proc. VLDB*. 524–525.

Grust, T., van Keulen, M., and Teubner, J. 2004. "Accelerating XPath evaluation in any RDBMS". *ACM Transactions on Database Systems* **29**, 91–131.

Gupta, A. K. and Suciu, D. 2003. "Stream Processing of XPath Queries with Predicates". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03).* 419–430.

Hidders, J. 2003. "Satisfiability of XPath Expressions". In *Proc. 9th DBPL*.

Hopcroft, J. E. and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Reading, MA USA.

Immerman, N. 1999. *"Descriptive Complexity".* Springer Graduate Texts in Computer Science.

Johnson, D. S. 1990. "A Catalog of Complexity Classes". In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. 1. Elsevier Science Publishers B.V., Chapter 2, 67–161.

Josifovski, V. and Fontoura, M. F. 2005. "Querying XML Streams". *VLDB Journal* **14,** 2 (April), 197–210.

Kamp, H. 1968. "Tense Logic and the Theory of Linear Order". Ph.D. thesis, University of California, Los Angeles.

Koch, C. 2003. "Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach". In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB).* 249–260.

Kolaitis, P. and Vardi, M. 2000. "Conjunctive Query Containment and Constraint Satisfaction". *Journal of Computer and System Sciences* **61,** 2, 302–332.

Lakshmanan, L. V. S., Ramesh, G., Wang, H., and Zhao, Z. 2004. "On Testing Satisfiability of Tree Pattern Queries". In *VLDB.* 120–131.

Lange, M. and Lutz, C. 2005. "2-ExpTime lower bounds for propositional dynamic logics with intersection". *"Journal of Symbolic Logic" 70,* 4, 1072–1086.

Libkin, L. 2004. *Elements of Finite Model Theory.* Springer.

Marx, M. 2004a. "Conditional XPath, the First Order Complete XPath Dialect". In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'04).* 13–22.

Marx, M. 2004b. "XPath with Conditional Axis Relations". In *Proc. EDBT.* 477–494.

Marx, M. 2005. "First order paths in ordered trees". In *Proc. of the 10th International Conference on Database Theory (ICDT).*

Marx, M. and de Rijke, M. 2004. "Semantic Characterizations of XPath". In *TDM'04 Workshop on XML Databases and Information Retrieval.* Twente, The Netherlands.

May, N., Brantner, M., Böhm, A., Kanne, C.-C., and Moerkotte, G. 2006. Index vs. navigation in xpath evaluation. In *Proc. Workshop Fourth International XML Database Symposium, Seoul, Korea.* 16–30.

Meyer, A. R. 1975. "Weak Monadic Second Order Theory of Successor is not Elementary-Recursive". In *Logic Colloquium, Lecture Notes in Mathematics 453.* Springer-Verlag, N.Y., 132–154.

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

Neumann, A. and Seidl, H. 1998. "Locating Matches of Tree Patterns in Forests". In *Proc. 18th FSTTCS, LNCS 1530.* 134–145.

Neven, F. 2002. "Automata Theory for XML Researchers". *SIGMOD Record* **31,** 3 (Sept.).

Neven, F. and Schwentick, T. 2002. "Query Automata on Finite Trees". *Theoretical Computer Science* **275**, 633–674.

Neven, F. and Schwentick, T. 2003. "XPath Containment in the Presence of Disjunction, DTDs, and Variables". In *Proc. of the 9th International Conference on Database Theory (ICDT).* 315–329.

Neven, F. and Van den Bussche, J. 2002. "Expressiveness of Structured Document Query Languages Based on Attribute Grammars". *Journal of the ACM* **49,** 1 (Jan.), 56–100.

Olteanu, D. 2007. "SPEX: Streamed and Progressive Evaluation of XPath". *IEEE Trans. Knowledge and Data Engineering* **19,** 7 (July).

Olteanu, D., Kiesling, T., and Bry, F. 5th - 8th March 2003. "An Evaluation of Regular Path Expressions with Qualifiers against XML Streams". In *Proceedings of 19th International Conference on Data Engineering (ICDE).* Bangalore, India. Full version in Technical Report PMS-FB-2002-12, Ludwig-Maximilians-Universität München, Munich, Germany, 2002.

Olteanu, D., Meuss, H., Furche, T., and Bry, F. 2002. "XPath: Looking Forward". In *Proc. EDBT Workshop on XML Data Management.* Vol. LNCS 2490. Springer-Verlag, Prague, Czech Republic, 109–127.

O'Neil, P. E., O'Neil, E. J., Pal, S., Cseri, I., Schaller, G., and Westbury, N. 2004. "ORDPATHs: Insert-Friendly XML Node Labels". 903–908.

Papadimitriou, C. H. 1994. *Computational Complexity.* Addison-Wesley.

Peng, F. and Chawathe, S. 2003. "XPath Queries on Streaming Data". In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03).*

RAMANAN, P. 2005. "Evaluating an XPath Query on a Streaming XML Document". In *Intl. Conf. Management of Data (COMAD)*. 41–52.

REINHARDT, K. 2002. "The Complexity of Translating Logic to Finite Automata". In *Automata, Logics, and Infinite Games – A Guide to Current Research*, E. Grädel, W. Thomas, and T. Wilke, Eds. Springer-Verlag, LNCS 2500.

SCHWENTICK, T. 2004. "XPath Query Containment". *SIGMOD Record 33,* 1, 101–109.

SCHWENTICK, T. 2007. "Automata for XML – A Survey". *Journal of Computer and Systems Science 73*, 289–315.

SCHWENTICK, T., THERÍEN, D., AND VOLLMER, H. 2001. "Partially-ordered Two-way Automata: A New Characterization of DA". In *Developments in Language Theory.* 239–250.

SEGOUFIN, L. AND VIANU, V. 2002. "Validating Streaming XML Documents". In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'02).*

SHANMUGASUNDARAM, J., TUFTE, K., ZHANG, C., HE, G., DEWITT, D. J., AND NAUGHTON, J. F. 1999. "Relational Databases for Querying XML Documents: Limitations and Opportunities". In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99).* 302–314.

SUDBOROUGH, I. 1977. "Time and Tape Bounded Auxiliary Pushdown Automata". In *Mathematical Foundations of Computer Science (MFCS'77).* Springer Verlag, LNCS 53, 493–503.

SUR, G., HAMMER, J., AND SIMEON, J. 2004. "An XQuery-Based Language for Processing Updates in XML". In *PLAN-X.*

TATARINOV, I., VIGLAS, S. D., BEYER, K., SHANMUGASUNDARAM, J., SHEKITA, E., AND ZHANG, C. 2002. "Storing and querying ordered XML using a relational database system". In *Proc. SIGMOD Conference.* 204–215.

THATCHER, J. AND WRIGHT, J. 1968. "Generalized Finite Automata Theory with an Application to a Decision Problem of Second-order Logic". *Mathematical Systems Theory* **2***,* 1, 57–81.

THÉRIEN, D. AND WILKE, T. 1998. "Over Two Variables Are as Powerful as One Quantifier Alternation: $FO^2 = \Sigma_2 \cap \Pi_2$". In *STOC.* 234–240.

VARDI, M. Y. 1982. "The Complexity of Relational Query Languages". In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC'82).* San Francisco, CA USA, 137–146.

VARDI, M. Y. 1995. "On the Complexity of Bounded-Variable Queries". In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'95).* San Jose, CA USA.

VARDI, M. Y. AND WOLPER, P. 1986. "Automata-theoretic techniques for Modal Logics of Programs". *Journal of Computer and System Sciences 32*, 183–221.

VENKATESWARAN, H. 1991. "Properties that Characterize LOGCFL". *Journal of Computer and System Sciences* **43**, 380–404.

WADLER, P. 2000. "Two Semantics for XPath". Draft paper available at http://www.research.avayalabs.com/user/wadler/.

WADLER, P. December 1999. "A Formal Semantics of Patterns in XSLT". In *Markup Technologies.* Philadelphia. Revised version in Markup Languages, MIT Press, June 2001.

WEIGEL, F., SCHULZ, K. U., AND MEUSS, H. 2005. "The BIRD Numbering Scheme for XML and Tree Databases - Deciding and Reconstructing Tree Relations Using Efficient Arithmetic Operations". In *Proc. XSym 2005.* 49–67.

WOOD, P. T. 2001. Minimizing Simple XPath Expressions. In *Proc. of Intl. Workshop on the Web and Databases (WebDB).* Santa Barbara, California, USA.

WOOD, P. T. 2003. "Containment for XPath Fragments under DTD constraints ". In *Proc. of the 9th International Conference on Database Theory (ICDT).* 300–314.

WORLD WIDE WEB CONSORTIUM. 1999a. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/.

WORLD WIDE WEB CONSORTIUM. 1999b. XSL Transformations (XSLT). W3C Recommendation Version 1.0.
http://www.w3.org/TR/xslt.

WORLD WIDE WEB CONSORTIUM. 2001. "XML Schema Part 0: Primer. W3C Recommendation". http://www.w3c.org/XML/Schema.

WORLD WIDE WEB CONSORTIUM. 2002. "XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft (Aug. 16th 2002). `http://www.w3.org/TR/query-algebra/`.

WORLD WIDE WEB CONSORTIUM. 2007. XML Path Language (XPath) 2.0.

YANNAKAKIS, M. 1981. "Algorithms for Acyclic Database Schemes". In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB'81).* Cannes, France, 82–94.