INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig-
Maximilians-
Universität
München

Evaluating Complex Queries against XML Streams with Polynomial Combined Complexity

Dan Olteanu, Tim Furche, François Bry

Technical Report, Computer Science Institute, Munich, Germany
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-2003-15, (Revised January 2004) 2003

Evaluating Complex Queries against XML Streams with Polynomial Combined Complexity

Dan Olteanu, Tim Furche, and François Bry

Institute for Informatics, University of Munich
{olteanu,timfu,bry}@pms.ifi.lmu.de

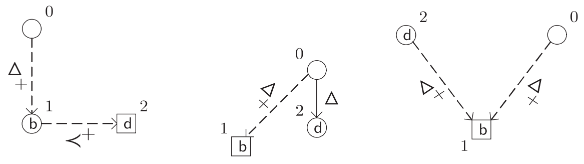
Abstract. Querying XML streams is receiving much attention due to its growing range of applications from traffic monitoring to routing of media streams. Existing approaches to querying XML streams consider restricted query language fragments, in most cases with exponential worst-case complexity in the size of the query. This paper gives correctness and complexity results for a query evaluator against XML streams called SPEX [8]. Its combined complexity is shown to be polynomial in the size of the data and the query. Extensive experimental evaluation with a prototype confirms the theoretical complexity results.

1 Introduction

Querying data streams is receiving an increasing attention due to emerging applications such as publish-subscribe systems, data monitoring applications like sensor networks [6], financial or traffic monitoring [3], and routing of media streams [10]. Such applications call for novel methods to evaluate complex queries against data streams. Data streams are preferred over data stored in memory for several reasons: (1) the data might be too large or volatile, or (2) a standard approach based on data parsing and storing might be too time consuming. For some applications, such as publish-subscribe systems or news distribution, streams of XML and semi-structured data are more appropriate than streams of relational data, as XML gives rise to trees with recursive structure definition and unbounded, yet finite depths. Recently, several approaches to querying streams of XML have been proposed, e.g., [1, 2, 5]. These approaches are restricted to rather weak query languages, but with efficient average-case complexity.

Contribution. This paper first reports on the query evaluator against XML streams called SPEX [8] and gives for it correctness and complexity results.

The query language used in the following, called RPQ, extends the XPath fragments for which streamed evaluations have been proposed, e.g., [1, 2, 5]. RPQ provides the core concepts of existing query languages for XML and provides support for the XPath axes child, descendant, following-sibling and their reverses parent, ancestor, and preceding-sibling, path, tree, and DAG queries. Although SPEX can process general DAG queries, this paper treats a restricted form of DAG queries, called single-join DAG queries, that can be efficiently evaluated.



(a) path (b) tree (c) DAG

Fig. 1. RPQ Queries

A single-join DAG is a DAG composed of two single-join DAGs or tree queries sharing at most one node. Many queries encountered in practice can be expressed as single-join DAG queries.

The time and space complexities for RPQ query evaluation with SPEX are shown to be polynomial in both the query and the stream size, and in fact near the theoretical optimum [4] for *in-memory* evaluation of an XPath fragment included in RPQ. Extensive experimental evaluation confirms the theoretical results and demonstrates almost constant memory use in practical cases.

The remainder of this paper is organized as follows. Section 2 recalls the stream data model and introduces RPQ. Section 3 presents the RPQ query evaluation by means of SPEX networks, and their complexity analysis is shaped in Section 4. Section 5 provides experimental results performed with a SPEX prototype on various streams and queries. Section 6 concludes the paper.

2 Preliminaries

XML Streams. SPEX [8] is designed to evaluate queries against XML streams conveying tree-shaped data with labels on nodes, where an a-node of the tree is represented in the stream by a pair of opening and closing XML tags $\langle \mathsf{a} \rangle$ and $\langle / \mathsf{a} \rangle$. For the sake of simplicity, only element nodes are supported.

Regular Path Queries (RPQ). For querying trees-shaped data, we use an abstraction of the navigational features of XPath called RPQ. The basic constructs of RPQ are binary relations. There are two base relations: the *child* relation $\lhd$ associating a node to its children, and the *next-sibling* relation $\prec$ associating a node to its immediate next sibling. For each base relation an inverse relation is defined, and for each base and inverse relation its transitive closure is defined as well. The grammar for full RPQ is specified next:

$$\mathit{RPQ} ::= \ \mathit{Identifier}(\mathit{Var}) \ \texttt{:-} \ \mathit{Expr}.$$

$$\mathit{Expr} ::= \ \mathit{Expr} \wedge \mathit{Expr} \ | \ \mathit{Expr} \vee \mathit{Expr} \ | \ (\mathit{Expr}) \ | \ \mathit{Var} \ \mathit{Relation} \ \mathit{Var} \ | \ \mathsf{Label}\,(\mathit{Var}).$$

$$\mathit{Relation} ::= \mathit{Base} \ | \ \mathit{Inverse} \ | \ \mathit{Base}^{+} \ | \ \mathit{Inverse}^{+}. \quad \mathit{Base} ::= \lhd \,|\, \prec . \quad \mathit{Inverse} ::= \rhd \,|\, \succ .$$

A relation expression $v\ r\ w$ associates two sets of nodes identified by the source variable v and the sink variable w that stand in relation r. Additionally, for each possible label there is a unary relation *Label* specifying the set of nodes with that label, e.g., $\mathsf{a}(v)$ restricts the set of nodes identified by v to nodes with label a. An RPQ query is an expression of the form $Q(t)$:- E where Q is an arbitrary identifier for the query, t is a variable occurring in E and E is an atomic expression such as $v_i \lhd v_j$ or $\mathsf{a}(v_i)$ or built up from atomic expressions using conjunctions or disjunctions. t is called the *head* variable, all other variables occurring in E are *body* variables.

Inverse relations are not explicitly considered in the following, for rewriting each expression $v\ \overline{r}\ w$, where $\overline{r}$ is the inverse to a base relation r, to $w\ r\ v$ yields an equivalent RPQ without that inverse relation. The equivalent RPQ is a single-join DAG query, where the variable v appears as sink of two expressions. Previous work of the authors [9] describes more sophisticated rewritings of queries with inverse relations yielding tree queries. Tree and single-join DAG queries are introduced below.

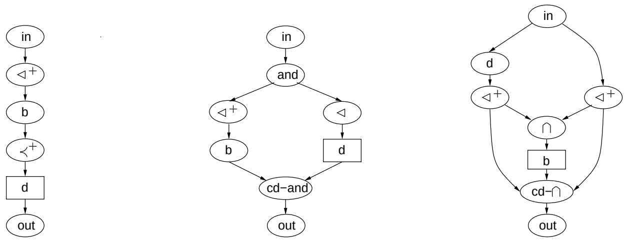
A *path query* is a query $Q(t)$:- E where in E (1) the head is the only non-source variable, (2) there is exactly one non-sink variable, (3) each variable occurs at most once as source and at most once as sink variable, and (4) there is no subset of atomic expressions such that source and sink of a conjunction of these atomic expressions are the same variable. RPQ paths correspond to XPath path expressions without predicates. Fig. 1(a) shows the path query $Q(v_2)$:- $v_0 \lhd^+ v_1 \wedge \mathsf{b}(v_1) \wedge v_1 \prec^+ v_2 \wedge \mathsf{d}(v_2)$.

A *tree query* is a path query $Q(t)$:- E where the first restriction is dropped and the third one is eased: each variable may occur in E at most once as sink but might occur several times as source of expressions. Hence, a tree query allows multi-source variables but no multi-sink variables. Tree queries correspond to XPath expressions with structural predicates. Fig. 1(b) shows the tree query $Q(v_1)$:- $v_0 \lhd^+ v_1 \wedge \mathsf{b}(v_1) \wedge v_0 \lhd v_2 \wedge \mathsf{d}(v_2)$, where v_0 is a multi-source variable.

A *DAG query* is a general query $Q(t)$:- E. A *single-join* DAG is a tree query where multi-sink variables are allowed and there are no two distinct paths in E with the same source *and* sink variable. Therefore, two distinct paths in E can share at most one variable. The single-join DAG query $Q(v_1)$:- $v_0 \lhd^+ v_1 \wedge \mathsf{b}(v_1) \wedge v_2 \lhd^+ v_1 \wedge \mathsf{d}(v_2)$, as depicted in Fig. 1(c), exemplifies a multi-sink variable v_1 occurring as sink in two relation expressions.

If only a source v_0 and a sink v_h variable from a subquery are of interest, such an expression can be abbreviated to $f(v_0, v_h)$, where f is an arbitrary identifier, e.g., $v_0 \lhd^+ v_1 \wedge \mathsf{b}(v_1) \wedge v_1 \prec^+ v_2 \wedge \mathsf{d}(v_2)$ can be abbreviated to $p(v_0, v_2)$.

The RPQ denotational semantics is given in the following by means of semantic mappings $\mathcal{R}$ and $\mathcal{S}$. Let n be the number of variables in an expression E and Nodes the set of nodes from a tree T conveyed by an XML stream. The set of all possible bindings for variables in E to nodes in Nodes, denoted Bindings, is the set of n-tuples Nodes^n, where each tuple contains one binding for each variable in E. For a tuple t, $t.v_i$ is the binding of the i-th variable in E. Given an expression E, the result of evaluating E against T is the subset $\mathcal{S}[\![E]\!](\beta)$ of $\beta = \text{Bindings}$. From the result set, the bindings for the head variable



(a) P-net (b) T-net (c) D-net

Fig. 2. Example of SPEX Networks

v can be obtained by a simple projection, as done by $\mathcal{R}$: for a query $Q(v)$:- E, $\mathcal{R}[\![Q(v)$:- $E]\!]$(Bindings) is the set of bindings of the head variable v to nodes in Nodes such that E holds. A relation expression $v\ r\ w$ retains only those tuples, where the binding for w is in $\mathcal{A}[\![r]\!](v)$. The function $\mathcal{A}$ maps each node in Nodes to the set of nodes in Nodes related to it under the relation r. The function $\mathcal{L}$ specifies for each label s the set of nodes in Nodes with that label.

$$\begin{aligned}
\mathcal{R} &: \text{Query} \rightarrow \text{Bindings} \rightarrow \text{Nodes}\\
\mathcal{R}[\![Q(v) \text{ :- } E]\!](\beta) &= \pi_v(\mathcal{S}[\![E]\!](\beta))\\
\mathcal{S} &: \text{Expression} \rightarrow \text{Bindings} \rightarrow \text{Bindings}\\
\mathcal{S}[\![E_1 \wedge E_2]\!](\beta) &= \mathcal{S}[\![E_1]\!](\beta) \cap \mathcal{S}[\![E_2]\!](\beta) = \mathcal{S}[\![E_1]\!](\mathcal{S}[\![E_2]\!](\beta)) = \mathcal{S}[\![E_2]\!](\mathcal{S}[\![E_1]\!](\beta))\\
\mathcal{S}[\![E_1 \vee E_2]\!](\beta) &= \mathcal{S}[\![E_1]\!](\beta) \cup \mathcal{S}[\![E_2]\!](\beta)\\
\mathcal{S}[\![(E)]\!](\beta) &= \mathcal{S}[\![E]\!](\beta)\\
\mathcal{S}[\![v_1\ r\ v_2]\!](\beta) &= \{t \in \beta \mid t.v_2 \in \mathcal{A}[\![r]\!](t.v_1)\}\\
\mathcal{S}[\![\mathsf{s}(v)]\!](\beta) &= \{t \in \beta \mid t.v \in \mathcal{L}[\![\mathsf{s}]\!]\}
\end{aligned}$$

Each RPQ expression is a filter applied to the current set of tuples containing variable bindings. Disjunctions and conjunctions of RPQ expressions can be directly mapped to unions and intersections of their operands' result. Conjunctions can also be expressed as the sequential application of $\mathcal{S}$ to the operands.

SPEX Transducer Networks. A single-state deterministic pushdown transducer is a tuple $(q, \Sigma, \Gamma, \delta)$, where q is the only state, Σ the input and output alphabet, Γ the stack alphabet, and the transition function δ is canonically extended to the configuration-based transition function $\vdash$: $\Sigma \times \Gamma^* \rightarrow \Gamma^* \times \Sigma^*$. E.g., the transition $(\langle a\rangle,\ [s] \mid \gamma) \vdash (\gamma,\ \langle a\rangle[s])$ reads: under the input symbol $\langle a\rangle$ and with $[s]$ as the top of the stack $|$-separated from the rest of the stack γ, the transducer outputs first the input symbol $\langle a\rangle$ *followed* by the top of the stack $[s]$ that is also popped from the stack.

Processing an XML stream with pushdown transducers corresponds to a depth-first traversal of the (implicit) tree conveyed by the XML stream. Exploiting the affinity between depth-first search and stack management, the transducers use their stacks for tracking the node depth in such trees. This way, RPQ relations can be evaluated in a single pass.

A SPEX *transducer network* is a directed acyclic graph where nodes are pushdown transducers. An edge between two transducers in a network inforces that an input tape of the sink transducer is an output tape of the source transducer. An evaluation plan for an RPQ query is represented in SPEX as a transducer network. Corresponding to the RPQ path, tree, and single-join DAG queries, there are three kinds of SPEX (transducer) networks: *P-nets* for the evaluation of path queries, *T-nets* for the evaluation of tree queries, and *D-nets* for the evaluation of single-join DAG queries.

3 The SPEX Evaluator

The evaluation of RPQ queries against XML streams with SPEX consists in two steps. First, an RPQ query is compiled into a SPEX transducer network. Second, the network computes the answers to the RPQ query.

The compilation of a query is defined inductively on the query structure: (1) Each RPQ relation is compiled into a pushdown transducer. (2) Each multi-source variable induces a pair of transducers (and, cd-and), if the variable is the source of conjuncted expressions, or (or,cd-or) for disjuncted expressions. These special transducers delimit the subnetworks corresponding to the expressions having that variable as source. (3) Each multi-sink variable induces a pair of transducers (∩,cd-∩), if the variable is the sink of conjuncted expressions, or (∪,cd-∪) for disjuncted expressions. (4) At the beginning of a network there is a stream-delivering in transducer and (5) at its end an answer-delivering out transducer. Fig. 2 shows networks for the RPQ queries of Fig. 1, where the *head* transducers that compute bindings for the head variable are marked with square boxes. Sections 3.1 through 3.4 detail the various transducers.

A network *net* for an expression $f(v_1, v_2)$ processes the XML stream enhanced by its transducer in with input bindings for v_1 and returns progressively output bindings for v_2, provided the expression holds. In the network, each transducer processes stepwise the received stream with input bindings and transmits it enhanced with computed output bindings to its successor transducers.

A node binding consists of a node annotated with a so-called *condition*. An annotation to a node is stored in the XML stream after the opening tag of that node. The output bindings delivered by a network *net* that processes the stream with some input bindings, contain the conditions of these input bindings.

The conditions are created by the in, and (or), and ∩ (∪) transducers. The in transducer binds *each non-sink* variable to all nodes, by annotating each node with (satisfied) conditions. In contrast, the RPQ semantics binds initially *each* variable to all nodes. Each and (or) transducer tests whether each received input binding satisfy the predicates represented by the subnetworks rooted at that and (or) transducer. In this sense, these transducers create an output binding with a new (satisfiable) condition, say $[s]$, for each received input binding, and sends it further to their successor subnetworks. The predicates are satisfied for that input binding (hence $[s]$ is satisfied), when $[s]$ is received by a cd-and (cd-or) within output bindings of each (at least one) subnetwork rooted at the corresponding and (or) transducer. The $\cap$ ($\cup$) transducer processes similarly to and (or).

We use in the following an abstraction of processing XML streams with a SPEX *net* for a query $Q(v_2)$ in terms of a function $out_Q(v_1)|_{v_2}$ that returns the bindings for v_2 constructed by *net*, when input bindings for v_1 are supplied. It is shown that (1) there is a corresponding $out_Q(v_1)|_{v_2}$ function that is actually implemented by *net*, and (2) there is an equality of $out_Q(v_1)|_{v_2}$ and $\mathcal{R}[\![Q(v_2)]\!]$.

3.1 Processing RPQ Relations

For a base relation r, a transducer $t(r)$ implements the function $out_r(v_1)|_{v_2} = \{v_2 | v_1\ r\ v_2\}$. Speaking in terms of annotations with conditions, if $t(r)$ receives a node n annotated with $[c]$ (hence a binding for v_1), then (1) it removes that annotation $[c]$ of n and stores it, (2) it identifies each node n' from the incoming stream that stands in relation r with n (hence n' is a binding for v_2), and (3) it annotates each such node n' with $[c]$. E.g., the child transducer $t(\lhd)$ annotates with $[c]$ all nodes n' that are children of n.

The equality of $out_r(v_1)|_{v_2}$ to RPQ semantics under bindings β follows by:

$$\mathcal{R}[\![Q(v_2)\ \texttt{:-}\ v_1\ r\ v_2]\!](\beta) = \pi_{v_2}(\mathcal{S}[\![v_1\ r\ v_2]\!](\beta)) = out_r(\pi_{v_1}(\beta))|_{v_2}.$$

A **condition** $[c]$, used to annotate nodes, follows immediately the opening tags of nodes, e.g., $\langle \mathsf{a} \rangle$[c]. Both, the opening tag $\langle \mathsf{a} \rangle$ and the condition $[c]$, represent a binding to that a-node. On the stacks of transducers and on the stream, conditions are expressed using (list of) integers, e.g., [1,2]. The operation $[c] \cup [s]$ denotes the set union of $[c]$ and $[s]$. There are two special conditions: the empty (unsatisfied) condition $[\,]$ and the *true* (satisfied) condition $[\top]$.

Configuration-based transitions defining the child $t(\lhd)$, descendant $t(\lhd^+)$, next-sibling $t(\prec)$, and next-siblings $t(\prec^+)$ transducers are given in the following. For all transducers, the input and output alphabet Σ consists of all opening $\langle x \rangle$ and closing $\langle /x \rangle$ tags and conditions $[c]$, and the stack alphabet Γ consists of all conditions $[c]$. Initially, an empty condition $[\,]$ is pushed on the stack of each transducer. The configurations of these transducers differ only in the first transition, which is actually a compaction of several simpler transitions that do only one stack operation. In transitions 2 and 3, x stands for any node label.

The *child* transducer $t(\lhd)$ is defined below. The transitions of this transducer read as follows: (1) if a condition $[c]$ is received, then $[c]$ is pushed on the stack and nothing is output; (2) if an opening tag $\langle x \rangle$ is received, then it is output followed by the condition from the top of the stack; (3) if a closing tag $\langle /x \rangle$ is received, then it is output and the top condition is popped from the stack.

$$\begin{array}{ll}
1. & ([c]\ ,\ \gamma) \vdash ([c] \mid \gamma,\ \varepsilon) \\
2. & (\langle x \rangle\ ,\ [s] \mid \gamma) \vdash ([s] \mid \gamma,\ \langle x \rangle [s]) \\
3. & (\langle /x \rangle,\ [s] \mid \gamma) \vdash (\gamma,\ \langle /x \rangle)
\end{array}$$

Lemma 1 ($t(\lhd)$ Correctness). *Transducer $t(\lhd)$ implements $out_{\lhd}(v_1)|_{v_2}$.*

Proof. When receiving a node n annotated with a condition $[c]$, $[c]$ is pushed on the stack. The following cases can then appear: (1) the closing tag of n is received, and $[c]$ is popped from the stack, for there are no other child nodes of n left in the incoming stream; (2) the opening tag of a child node n' of n is received, and it is output followed by $[c]$ (hence child nodes n' are annotated correctly with $[c]$). In the latter case another condition $[c']$ is received afterwards, pushed on the stack, and used to annotate child nodes of n'. Only when the closing tag of n' is received, $[c']$ is popped and $[c]$ becomes again the top of the stack. At this time, siblings of n' can be received and annotated with $[c]$ (the above case 2), or the closing tag of n is received (the above case 1). □

The *descendant* transducer $t(\lhd^+)$ is defined below. The missing transitions 2 and 3 are like for $t(\lhd)$. In the first transition, $t(\lhd^+)$ pushes on the stack the received condition $[c]$ *together* with the top condition $[s]$: $[c]\cup[s]$. The difference to $t(\lhd)$ is that also conditions that annotate the ancestors n_a of n are used to annotate child nodes n' of n, for the nodes n' are descendants of nodes n_a.

$$1.\ ([c],\ [s] \mid \gamma) \vdash ([c]\cup[s] \mid [s] \mid \gamma,\ \varepsilon)$$

Lemma 2 **($t(\lhd^+)$ Correctness).** *Transducer $t(\lhd^+)$ implements $out_{\lhd^+}(v_1)|_{v_2}$.*

Proof. When receiving a node n annotated with a condition $[c]$, $[c]$ is pushed on the stack together with the current top $[s]$: $[c]\cup[s]$. The following cases can then appear: (1) the closing tag of n is received, and $[c]\cup[s]$ is popped from the stack, as there are no other descendants of n left in the incoming stream; (2) the opening tag of a child node n' of n is received, and it is output followed by $[c]\cup[s]$ (hence descendant nodes n' are annotated correctly). In the latter case another condition $[c']$ is received afterwards, the condition $[c']\cup[c]\cup[s]$ is pushed on the stack and used to annotate child nodes of n'. Only when the closing tag of n' is received, $[c']\cup[c]\cup[s]$ is popped and $[c]\cup[s]$ becomes again the top of the stack. At this time, siblings of n' can be received and annotated with $[c]\cup[s]$ (the above case 2), or the closing tag of n is received (the above case 1). □

The *next-sibling* transducer $t(\prec)$ is defined below. The missing transitions 2 and 3 are like for $t(\lhd)$. In the first transition, $t(\prec)$ replaces the top of the stack $[s]$ with the received condition $[c]$ and pushes an empty condition $[\,]$. The condition $[\,]$ is then used to annotate child nodes n' of the received node n annotated with $[c]$. When the closing tag of n is received, the condition $[\,]$ is popped and the next sibling node of n is annotated with the top condition $[c]$.

$$1.\ ([c],\ [s] \mid \gamma) \vdash ([\,] \mid [c] \mid \gamma,\ \varepsilon)$$

The *next-siblings* transducer $t(\prec^+)$ is defined below. The missing transitions 2 and 3 are like for $t(\lhd)$. In the first transition, $t(\prec^+)$ adds to the top of the stack $[s]$ the received condition $[c]$ that annotated a node n, and pushes a condition $[\,]$. The difference to $t(\prec)$ is that the top of the stack $[s]$ is kept together with the received condition $[c]$: $[c]\cup[s]$. In this way, conditions that annotated the preceding siblings of n (like $[s]$) are used to annotate the following siblings of n.

1. $([c],\ [s] \mid \gamma) \vdash ([\,] \mid [c]{\cup}[s] \mid \gamma,\ \varepsilon)$

Analogous to Lemmas 1 and 2, the $t(\prec)$ transducer implements $out_{\prec}(v_1)|_{v_2}$ and the $t(\prec^+)$ transducer implements $out_{\prec^+}(v_1)|_{v_2}$.

The *label* transducer $t(\mathsf{a})$ for a label a acts like a binding filter: the input bindings with node labels *matched* by the transducer parameter a are output, the other ones are filtered out, i.e., their conditions are replaced with the empty condition $[\,]$. The label matching can be here extended to regular expression matching. It is easy to see that $\mathcal{R}[\![Q(v_1)\ \texttt{:-}\ \mathsf{a}(v_1)]\!](\beta) = out_{\mathsf{a}}(\pi_{v_1}(\beta))|_{v_1}$.

3.2 Processing Path Queries

A path query is a sequence of relations such that for two consecutive relations there is a unique variable that is the sink of the first and the source of the second relation. SPEX compiles a path query into a P-net that is a sequence of connected transducers. A connection between two transducers consists in having the second transducer processing the output of the first one. The answers computed by a P-net are the nodes annotated by its last but one transducer. Fig. 2(a) shows the P-net for the query $P(v_2)\ \texttt{:-}\ v_0 \lhd^+ v_1 \wedge \mathsf{b}(v_1) \wedge v_1 \prec^+ v_2 \wedge \mathsf{d}(v_2)$ from Fig. 1(a) and Example 1 shows stepwise its processing.

The P-net $t(P) = [t(r_1), \ldots, t(r_n)]$ for a general path query $P(v_n)\ \texttt{:-}\ v_0\ r_1\ v_1 \wedge \ldots \wedge v_{n-1}\ r_n\ v_n$ is an implementation of $out_P(v_0)|_{v_n} = out_{r_n}(..(out_{r_1}(v_0)|_{v_1})..)|_{v_n}$: for given input bindings v_0, out_P returns output bindings v_n, such that for every $1 \leq i \leq n$ the relation $v_{i-1}\ r_i\ v_i$ holds. For a set of bindings β, it follows that:

$$
\begin{aligned}
&\mathcal{R}[\![P(v_n)\ \texttt{:-}\ v_0\ r_1\ v_1 \wedge \ldots \wedge v_{n-1}\ r_n\ v_n]\!](\beta) = \\
&\quad = \pi_{v_n}(\mathcal{S}[\![v_{n-1}\ r_n\ v_n]\!](\ldots(\mathcal{S}[\![v_0\ r_1\ v_1]\!](\beta))\ldots)) \\
&\quad = out_{r_n}(\ldots(out_{r_2}(\pi_{v_1}(\mathcal{S}[\![v_0\ r_1\ v_1]\!](\beta)))|_{v_2})\ldots)|_{v_n} \\
&\quad = out_{r_n}(\ldots(out_{r_2}(\pi_{v_1}(out_{r_1}(\pi_{v_0}(\beta))|_{v_1}))|_{v_2})\ldots)|_{v_n} \\
&\quad = out_{r_n}(\ldots(out_{r_1}(\pi_{v_0}(\beta))|_{v_1})\ldots)|_{v_n} = out_P(\pi_{v_0}(\beta))|_{v_n}.
\end{aligned}
$$

The correctness of a P-net $t(P)$ implementation of $out_P(v_0)|_{v_n}$, where $t(r_i)$ implements $out_{r_i}(v_{i-1})|_{v_i}$, $(1 \leq i \leq n)$, follows from the observation that the output of each transducer from P-net is streamed in the next transducer.

Example 1. Consider the path query $P(v_2)\ \texttt{:-}\ v_0 \lhd^+ v_1 \wedge \mathsf{b}(v_1) \wedge v_1 \prec^+ v_2 \wedge \mathsf{d}(v_2)$ from Fig. 1(a), which selects the next-siblings d-nodes of every b-node. The corresponding P-net is $t(P) = [t(\lhd^+), t(\mathsf{b}), t(\prec^+), t(\mathsf{d})]$, as shown in Fig. 2(a). Table 1 gives a stream fragment *in* annotated with conditions, the intermediate and result output streams generated by the transducers in the P-net $t(P)$. Note that the intermediary output streams are created progressively and *not* stored. Thus, following the evaluation process corresponds to reading entire columns from left to right.

Recall that the stack of each transducer is initialized with the empty condition $[\,]$. The transducers do the following stack and output operations for processing the first two opening tags and two conditions:

in	⟨a⟩[1]	⟨b⟩[2]	⟨b⟩[3]	⟨/b⟩	⟨/b⟩	⟨d⟩[4]	⟨/d⟩	⟨/a⟩
$out_{\lhd^+}$	⟨a⟩[ ]	⟨b⟩[1]	⟨b⟩[1,2]	⟨/b⟩	⟨/b⟩	⟨d⟩[1]	⟨/d⟩	⟨/a⟩
out_{b}	⟨a⟩[ ]	⟨b⟩[1]	⟨b⟩[1,2]	⟨/b⟩	⟨/b⟩	⟨d⟩[ ]	⟨/d⟩	⟨/a⟩
$out_{\prec^+}$	⟨a⟩[ ]	⟨b⟩[ ]	⟨b⟩[ ]	⟨/b⟩	⟨/b⟩	⟨d⟩[1]	⟨/d⟩	⟨/a⟩
out_{d}	⟨a⟩[ ]	⟨b⟩[ ]	⟨b⟩[ ]	⟨/b⟩	⟨/b⟩	⟨d⟩[1]	⟨/d⟩	⟨/a⟩

Table 1. Processing Example with P-net

On receiving ⟨a⟩[1], the first transducer in $t(P)$, i.e., $t(\lhd^+)$, outputs ⟨a⟩ and its top condition [ ], and pushes the received condition [1]. The next transducer $t(b)$ receives ⟨a⟩[ ] and sends it further to $t(\prec^+)$, which outputs ⟨a⟩ followed by its top condition [ ], adds later on the received condition [ ] to the top condition [ ], and pushes an empty condition [ ]. The next transducer $t(d)$ receives then ⟨a⟩[ ] and sends it further.

On receiving ⟨b⟩[2], $t(\lhd^+)$ outputs ⟨b⟩ and its top condition [1], and pushes [1,2]. Next, $t(b)$ receives ⟨b⟩[1] and sends it further to $t(\prec^+)$, which outputs ⟨b⟩ followed by its top condition [ ], adds later on the received condition [1] to the top condition [ ], and pushes an empty condition [ ]. The next transducer $t(d)$ receives then ⟨b⟩[ ] and sends it further.

3.3 Processing Tree Queries

A tree query is a path query extended with multi-source variables. There is one path in a tree query that leads from a non-sink variable to the head variable, called the *head* path, the other non-head paths are called *predicates*. SPEX compiles a tree query into a T-net. Each multi-source variable introduces a pair of special transducers (and, cd-and), if that variable is source of conjuncted expressions, or (or,cd-or) for disjuncted expressions. These transducers delimit the subnetworks corresponding to the expressions having that variable as source. Fig. 2(b) shows the T-net for the query $T(v_1)$:- $v_0 \lhd^+ v_1 \wedge \mathsf{b}(v_1) \wedge v_0 \lhd v_2 \wedge \mathsf{d}(v_2)$ from Fig. 1(b).

The answers computed by a T-net are among the nodes annotated by its head. These nodes are *potential* answers, as they may depend on a downstream satisfaction of T-net predicates. The predicate satisfaction is conveyed in the T-net by conditions that annotate nodes. Until the predicate satisfaction is decided, the potential answers are buffered by the last transducer out in the T-net. Consider, e.g., the evaluation of the tree query $T(v_1)$:- $v_0 \lhd^+ v_1 \wedge \mathsf{a}(v_1) \wedge v_1 \lhd v_2 \wedge \mathsf{b}(v_2)$. When encountering in the XML stream an opening tag ⟨a⟩ marking the beginning of an a-node, it is not yet known whether this node has a b-node child, i.e., whether it is an answer or not. Indeed, by definition of XML streams, such a child can only appear downstream. This might remain unknown until the corresponding closing tag ⟨/a⟩ is processed. At this point, it is impossible for the a-node to have further b-node children. Thus, the stream fragment corresponding to a potential answer has to be buffered as long as it is not known whether predicates that might apply are satisfied or not, but no longer.

A non-empty condition $[c]$ annotating a node n is replaced by an and (or) transducer with a new condition $[q]$, where q is the stack size of that transducer. The transducer also pushes $[q]$ on its stack, and forwards to its condition determinant transducer cd (cd-and or cd-or) the condition mapping $[c]{\rightarrow}[q]$. Each subnetwork routed at that and (or) transducer receives $[q]$ and when $[q]$ is received from all (at least one) ingoing edges of the cd-and (cd-or), $[q]$ is considered satisfied. Using the condition mapping $[c]{\rightarrow}[q]$, the cd transducer forwards $[c]$ to the cd transducer corresponding to the preceding and (or) transducer that created $[c]$, or of the out transducer corresponding to the in transducer. However, as soon as it is known that $[q]$ can no longer be satisfied, $[q]$ is considered unsatisfied and the nodes annotated with $[q]$ by the head and buffered by the out transducer are discarded.

The condition mapping $[c]{\rightarrow}[q]$ is discarded when the cd transducer receives (1) the closing tag of n, or (2) the closing tag of the parent node of n. The former applies for and/or transducers followed by subnetworks that start with $t(\lhd)$ or $t(\lhd^+)$, as $t(\lhd)$ or $t(\lhd^+)$ and their subsequents in the subnetworks can create output bindings with $[q]$ only within the subtree rooted by the node n (and hence enclosed within the opening and closing tags of n). The latter applies for and/or transducers followed by subnetworks that start with $t(\prec)$ or $t(\prec^+)$, as they can create output bindings with $[q]$ only within the stream fragment starting with the closing tag of n and ending with the closing tag of the parent of n. The lifetime of a condition mapping, i.e., the time between its creation and its discarding, influences the number of condition mappings alive at a time. In the former above case, there can be at most d condition mappings alive at a time, where d is the depth of the tree conveyed by the input stream, whereas in the latter above case, there can be at most $d + b$ condition mappings alive at a time, where b is the breadth of the tree conveyed by the input stream.

Condition mappings are indispensable for representing condition scopes in the network's computation. A network for a query with p multi-source variables has p (and/or, cd) pairs, hence p condition scopes. Consider the condition mappings $[c_i]{\rightarrow}[c_{i+1}]$ $(1 \leq i \leq p)$ created by a transducer network with p condition scopes, where each mapping corresponds to a scope. If the head has annotated nodes with $[c_h]$, then they become answers only when $[c_h]$ is satisfied and from each other scope i $(1 \leq i \leq p, i \neq h)$ at least one condition $[c_i]$ that is mapped directly or indirectly to $[c_h]$ is also satisfied. As soon as they become answers, they are output and removed from the buffer.

Let us consider the correctness of T-nets for tree queries with conjunctions. For tree queries with disjunctions similar treatment can be applied. A tree query $T(v_h)$:- $h(v_0, v_h) \wedge q_i(v_j, v_i)$ with path $h(v_0, v_h)$, which leads to the head variable v_h via intermediate variables v_j $(0 \leq j \leq h)$, and predicates $q_i(v_j, v_i)$ $(h < i \leq m)$ is compiled into a T-net network $t(T) = (t(h), t(q_i))$ with a head P-net $t(h)$ and predicate P-nets $t(q_i)$. The T-net $t(b)$ is an implementation of a function $out_b(v_0)|_{v_h} = \{out_h(v_0)|_{v_h} \mid \forall h < i \leq m, \exists v_i \in out_{q_i}(v_j)|_{v_i}, 0 \leq j \leq h\}$: for given input bindings v_0, it returns output bindings for v_h, such that the path $h(v_0, v_h)$ holds and there exists a binding of v_i for each $q_i(v_j, v_i)$. It follows that:

$$\begin{aligned}
&\mathcal{R}[\![T(v_h) \text{ :- } h(v_0, v_h) \land q_i(v_j, v_i)]\!](\beta) = \pi_{v_h}(\mathcal{S}[\![h(v_0, v_h)]\!](\beta) \cap \mathcal{S}[\![q_i(v_j, v_i)]\!](\beta)) \\
&= \pi_{v_h}(\{t \mid t \in \mathcal{S}[\![h(v_0, v_h)]\!](\beta), t \in \mathcal{S}[\![q_i(v_j, v_i)]\!](\beta)\}) \\
&= \{t.v_h \mid t.v_h \in out_h(\pi_{v_0}(\beta))|_{v_h}, \exists t.v_i \in out_{q_i}(\pi_{v_j}(\beta))|_{v_i}\} = out_T(\pi_{v_0}(\beta))|_{v_h}.
\end{aligned}$$

The correctness of a T-net $t(T) = (t(h), t(q_i))$ implementation of $out_T(v_0)|_{v_h}$, where h and q_i implement $out_h(v_j)|_{v_h}$ and $out_{q_i}(v_0)|_{v_i}$ ($h < i \leq m$,$0 \leq j \leq h$) follows from the above characterizations of and and cd-and.

3.4 Processing Single-Join DAG Queries

A single-join DAG query consists of several subqueries that share only one variable. SPEX compiles such queries into D-nets. Each multi-sink variable introduces a pair of special transducers ($\cup$, cd-$\cup$), if the variable is the sink of disjuncted expressions, or ($\cap$, cd-$\cap$) for conjuncted expressions. Fig. 2(c) shows the D-net for the query $D(v_1)$:- $v_0 \lhd^+ v_1 \land \mathsf{b}(v_1) \land v_2 \lhd^+ v_1 \land \mathsf{d}(v_2)$ from Fig. 1(c).

The transducers $\cap/\cup$ are similar to the and/or transducers. However, as set operations are defined for $k \geq 2$ operands, these transducers have k ingoing edges and their cd transducers have $k + 1$ incoming edges, one edge for each subnetwork implementing an operand and one edge for the subnetwork enclosed. For each node, a set transducer receives also a condition $[c_i]$ $(1 \leq i \leq k)$ from each ingoing edge and possibly creates a mapping $[c_i]{\to}[q]$. The $\cap/\cup$ transducer creates the new condition $[q]$ only if all (at least one) $[c_i]$ conditions are non-empty. If potential answers are annotated with $[q']$, then they become answers only when $[q']$ is satisfied, one condition that is mapped directly or indirectly to $[q']$ from each condition scope is satisfied, and for the $\cap/\cup$ transducer all (at least one) $[c_i]$ conditions are satisfied.

The correctness of D-nets can be proven similarly to P-nets and T-nets.

4 Analytical Complexity

The evaluation of RPQ with SPEX has a polynomial combined complexity in the stream and the query size, near the optimum [4] for *in-memory* evaluation of the XPath fragment included in RPQ. We assume that the tree conveyed by the XML stream has depth d, breadth b, and size s. We define four RPQ classes: RPQ_1 contains path queries and their conjunctions and disjunctions, RPQ_2 contains queries without closure relations, RPQ_3 contains RPQ_2 and the $\lhd^+$ relation, and RPQ_4 contains RPQ_3 and the $\prec^+$ relation.

Theorem 1. *The time T_i and space S_i complexities for processing RPQ_i are:*

1. $T_1 = O(q \times s)$ *and* $S_1 = O(q \times d)$.
2. $T_2 = O(q \times s)$ *and* $S_2 = O(q \times d + s)$.
3. $T_3 = O(q \times d \times s)$ *and* $S_3 = O(q \times d^2 + s)$.

4. $T_4 = O(q \times \max(d, b) \times s)$ *and* $S_4 = O(q \times d \times \max(d, b) + s)$.

Proof. The size of a network for a RPQ query is linear in the size of the query. A stack can have d entries, for every opening tag brings a condition that is pushed on the stack, and its corresponding closing tag pops a condition. For evaluating queries with multi-source variables, i.e., with predicates, the extra space s can be needed for buffering potential answers. This buffering is independent of SPEX and in some cases unavoidable. The entire space s is needed only in pathological cases, e.g., when the entire XML stream is a potential answer that depends on a condition satisfaction which can be decided only at the end of the XML stream.

(1) In a network for a query without multi-source variables, only the in transducer creates new conditions, which are in fact satisfied conditions $[\top]$. Hence, the condition unions done by transducers for closure relations yield always satisfied conditions of constant size. Thus, the entries on any transducer stack have constant size. The time needed to read/write a tag or condition is constant.

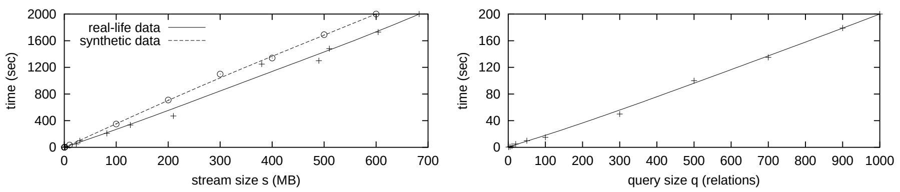
(2) No closure relations in the query means no transducers in the network to compute unions of conditions and consequently conditions of constant size. Processing queries with multi-source variables can require extra space s.

(3) The $t(\lhd^+)$ transducer computes condition unions. As there can be d condition mappings created at a time within a condition scope and stored on stacks, a condition union has maximum size d. The stacks have d entries, hence the size of a stack can be d^2. To read/write a condition can take d time.

(4) The $t(\prec^+)$ transducer computes condition unions. An and/or can store $b+d$ condition mappings at a time, if a $t(\prec^+)$ immediately follows it. Otherwise, case (3) applies. A condition can have $\max(d, b)$ size. □

The extra space s that can add to the worst-case space complexity of processing queries of RPQ_i ($i \geq 2$) classes with SPEX is not necessary for the evaluation of queries with multi-source variables that have their predicates always being evaluated before the head transducer annotates potential answers. In this case, it is already known whether the conditions used by the head transducer for annotation are satisfied. E.g., in the evaluation of the query $T(v_2)$:- $v_0 \lhd^+ v_1 \wedge \mathsf{b}(v_1) \wedge v_0 \prec^+ v_2 \wedge \mathsf{d}(v_2)$, the predicate $b(v_1)$:- $v_0 \lhd^+ v_1 \wedge \mathsf{b}(v_1)$ is always evaluated before the head path $h(v_2)$:- $v_0 \prec^+ v_2 \wedge \mathsf{d}(v_2)$, because the nodes bound to v_1 are among the descendants of the nodes bound to v_0, whereas the nodes bound to v_2 are among the following siblings of the nodes bound to v_0 and therefore are encountered later in the stream. The class RPQ_5 of queries, the evaluation of which does not require to buffer stream fragments:

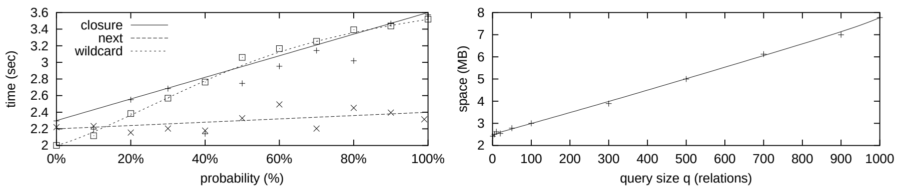
$$
\begin{aligned}
RPQ_5 = RPQ_1 \cup \{&Q(v_1) \text{ :- } f_0(v_0, x) \wedge f_1(x, v_1) \wedge f_i(x, v_i) \mid \\
&f_1(x, v_1) = x\ (\prec | \prec^+)\ v_1' \wedge f_1'(v_1', v_1), f_i(x, v_i) = x\ (\lhd | \lhd^+)\ v_i' \wedge f_i'(v_i', v_i), \\
&f_0(v_0, x) \in RPQ_5, f_1'(v_1', v_1) \in RPQ_5, f_i'(v_i', v_i) \in RPQ, i \geq 2\}.
\end{aligned}
$$



(a) Varying stream size s ($q = 10$, $3 \leq d \leq 32$)

(b) Varying query size q ($s = 244$ kB, $d = 32$)

Fig. 3. Scalability ($p_* = p_+ = p_\prec = p_\curlywedge = p_\curlyvee = 0.5$)



(a) Effect of p_*, p_+, and $p_\prec$

(b) Effect of varying query size q

Fig. 4. If not varied, $s = 244$ kB, $d = 32$, $q = 10$, $p_* = p_+ = p_\prec = p_\curlywedge = p_\curlyvee = 0.5$

5 Experimental Evaluation

The theoretical results of Section 4 are verified by an extensive experimental evaluation conducted on a prototype implementation of SPEX in Java (Sun Hotspot JRE 1.4.1) on a Pentium 1.5 GHz with 500 MB under Linux 2.4.

XML Streams. The effect of varying the stream size s on evaluation time is considered for two XML stream sets. The first set [7] provides real-life XML streams, ranging in size from 21 to 21 million elements and in depth from 3 to 36. The second set provides synthetic XML streams with a slightly more complex structure that allows more precise variations in the workload parameters. The synthetic data is generated from information about the currently running processes on computer networks and allows the specification of both the size and the maximum depth of the generated data.

Queries. Only RPQ queries that are "schema-aware" are considered, i.e., that express structures compatible with the schema of the XML streams considered. Their generation has been tuned with the query size q and several probabilities: $p_\prec$ and p_+ for next-sibling, resp. closure relations, $p_\curlywedge$ and $p_\curlyvee$ for a branch, resp. a join, and p_* for the probability that a variable has a label relation. E.g., a path query has $p_\curlywedge = p_\curlyvee = 0$ and a tree query $p_\curlyvee = 0$. For each parameter setting, 10–30 queries have been tested , totaling about 1200 queries.

Scalability. Scalability results are only presented for stream and query size. In all cases, the depth is bounded in a rather small constant ($d \leq 36$) and its influence on processing time showed to be considerably smaller than of the stream and query size. Fig. 3 emphasizes the theoretical results: Query processing time increases linearly with the stream size as well as with the query size. The effect is visible in both the real-life and the synthetic data set, with a slightly higher increase in the synthetic data due to the more complex structure.

Varying the query characteristics. Fig. 4(a) shows an increase of the evaluation time by a factor of less than 2 when p_* and p_+ increase from 0 to 100%. It also suggests that the evaluation times for $\prec$ and $\lhd$ are comparable. Further experiments have shown that the evaluation of tree and DAG queries operations is slightly more expensive than the evaluation of simple path queries.

The **memory usage** is almost constant over the full range of the previous tests. Cf. Fig. 4(b), an increase of the query size q from 1 to 1000 leads to an increase from 2 to 8 MB of the memory for the network and for its processing. The memory use is measured by inspecting the properties of the Java virtual machine (e.g., using `Runtime.totalMemory()` and `Runtime.freeMemory()`).

6 Conclusion

This paper gives correctness and complexity results for the SPEX [8] query evaluator against XML streams. SPEX evaluates XPath-like queries, i.e., path, tree, and single-join DAG queries, with polynomial time and space complexity. The complexity results are confirmed by extensive experimental evaluation.

Acknowledgments. We thank the anonymous reviewers and Holger Meuss that made many helpful suggestions on the penultimate draft.

References

1. M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of VLDB*, pages 53–64, 2000.
2. C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. of ICDE*, pages 235–244, 2002.
3. Cisco Systems. Cisco IOS netflow, 2000. `http://www.cisco.com/warp/public/cc/pd/iosw/prodlit/iosnf_ds.pdf`.
4. G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. of PODS*, pages 179–190, 2003.
5. T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proc. of ICDT*, pages 173–189, 2003.
6. S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of ICDE*, pages 555–566, 2002.
7. G. Miklau. XMLData repository, Univ. of Washington, 2003. `http://www.cs.washington.edu/research/xmldatasets`.
8. D. Olteanu, T. Furche, and F. Bry. An Efficient Single-Pass Query Evaluator for XML Data Streams. In *Proc. of ACM SAC*, pages 627–631, 2004.
9. D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. of EDBT Workshop XMLDM*, pages 109–127, 2002. LNCS 2490.
10. D. Rogers, J. Hunter, and D. Kosovic. The TV-trawler project. *J. of Imaging Systems and Technology*, pages 289–296, 2003.