SPEX: Streamed and Progressive Evaluation of XPath

Dan Olteanu

***Abstract*— Streams are preferable over data stored in memory in contexts where data is too large or volatile, or a standard approach to data processing based on storing is too time or space consuming. Emerging applications such as publish-subscribe systems, data monitoring in sensor networks, financial and traffic monitoring, and routing of MPEG-7 call for querying streams. In many such applications, XML streams are arguably more appropriate than flat streams, for they convey (possibly unbounded) unranked ordered trees with labeled nodes. However, the flexibility enabled by XML streams in data modeling makes query evaluation different from traditional settings and challenging.**

This article describes SPEX, a streamed and progressive evaluation of XPath. SPEX compiles queries into networks of simple and independent transducers and processes XML streams with polynomial combined complexity. This makes SPEX especially suitable for implementation on devices with low-memory and simple logic as used, e.g., in mobile computing.

***Index Terms*— Query Evaluation, Streams, Transducers, XML, XPath**

I. Introduction

STREAMS are an emerging technology for data dissemination in cases where the data throughput or size make it unfeasible to rely on the conventional approach based on storing the data before processing it [1]. Areas where streams are applied include monitoring of scientific data (environments [2], astronomy [3], meteorology), control data (traffic [4], networks [5], logistics), financial data (bank transactions [6]), and MPEG-7 routing [7]. Streams are complementary and symmetrical to traditional databases. While in traditional databases data is persistent and queries are volatile, in stream applications data is volatile but queries are persistent. Streams are a new and promising setting in which many conventional database methods have to be considered anew.

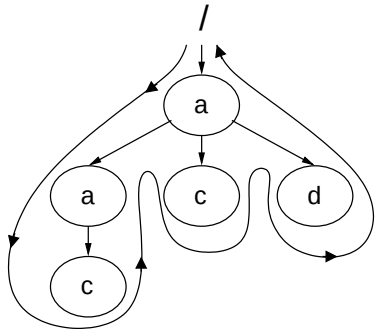
Querying XML streams without storing and without decreasing considerably the data throughput is especially challenging because XML streams can convey tree structured data with unbounded size and depth. Important desiderata for query processors against XML streams are to employ streamed and progressive evaluation, to scale in both data and query size, and to offer support for reasonably expressive query languages. Streamed evaluation means here that only one pass over the XML stream is used, and progressive evaluation means that the answers are output as soon as possible. Current streamed query processors, e.g., [8]–[14], are not designed to accomplish all these desiderata, their focus being in most cases a subset of them, possibly with additional desiderata, e.g., support for indexing and evaluating large sets of simple queries. We survey these processors in Section IX.

Dan Olteanu is with the Department of Informatics, Saarland University, 66041 Saarbrücken, Germany. E-mail: olteanu@infosys.uni-sb.de

The main contribution of this article is an evaluation method that fulfills all four above desiderata. More precisely, the contributions are as follows.

- We describe a streamed and progressive query evaluation against XML streams (SPEX for short). Extended abstracts on SPEX are given by [15]–[17].
- The query language supported by SPEX is Forward Core XPath [18] extended with path union and path difference. This is a clean fragment of XPath [19]. XPath lies at the core of important languages for the Web, e.g., the query language XQuery [20], the transformation language XSLT [21], the schema language XML-Schema [22], and the language for addressing fragments of XML documents XPointer [23].
- SPEX has polynomial combined complexity, i.e., polynomial in *both* the data and the query sizes. Chronologically, SPEX is the first streamed Core XPath processor to enjoy polynomially combined complexity [16]. This contrasts with most approaches to streamed evaluation, which have exponential query complexity, e.g., [8]–[11].
- We show that SPEX is scalable. Our experiments show that SPEX scales for queries of 1000 steps as well as real-life XML documents of 700 MB (the biggest sizes in our experiments). Further experiments confirm SPEX scalability for application-generated XML streams [17].
- SPEX is extensible by design. It compiles queries into networks of independent transducers and the addition of transducers implementing new query constructs does not influence the behavior of the existing ones. Note that the aforementioned streamed XPath processors are not extensible, for they are specifically designed for very small XPath fragments.
- To further improve the evaluation time and also to give an example of useful SPEX extensions, we introduce so-called filters that reduce the stream traffic within transducer networks. We experimentally confirm that the filters are very effective, especially for networks representing selective queries.
- SPEX is an open-source processor publicly available at `http://spex.sourceforge.net`. The prototype implements also comparisons with constants and a restricted form of aggregation (count), which are not discussed here.

We proceed as follows. In Section II we define annotated XML streams and introduce the XPath fragment of concern. Section III overviews the main ideas of SPEX. We show how XPath queries are compiled into transducer networks in Section IV and define various transducers in Section V. An optimization technique for reducing the stream traffic within networks is described in Section VI. The complexity study of SPEX follows in Section VII and experiments are reported in Section VIII. Finally, we give credits to related work in Section IX and conclude in Section X.



⟨⟩[0] ⟨a⟩[ ] ⟨a⟩[ ] ⟨c⟩[ ] ⟨/c⟩⟨/a⟩⟨c⟩[ ] ⟨/c⟩⟨d⟩[ ] ⟨/d⟩⟨/a⟩⟨/⟩

Fig. 1. A tree and its corresponding XML stream (⟨⟩stands for stream beginning and ⟨/⟩for stream end).

II. Preliminaries

A. Annotated XML Streams

XML streams correspond to depth-first, left-to-right, preorder serializations of trees. Each node is represented by an opening and closing tag as follows: on entering (exiting) that node, its opening (closing) tag is appended to the stream. For our purpose, each opening tag is followed by an annotation, cf. Fig.1. Annotations are used to mark selected nodes during query evaluation. The annotations of particular nodes of interest are marked with a *head* flag. This is the case of the nodes that can be in the answer (so-called answer candidates).

An annotation is expressed as a list of positive integers in ascending order, e.g., [1,2]. There are two special annotations: the empty annotation, noted [ ], corresponding to the empty list, and the full annotation, noted [0], corresponding to the list containing all positive integers. There are three operations defined for annotations: union $\sqcup$, intersect $\sqcap$, and inclusion $\sqsubseteq$, whose semantics resemble that of standard set operations $\cup$, $\cap$, and $\subseteq$. For example, the operation $c \sqcup s$ denotes the union of annotations c and s with duplicate removal, like in [1,2]$\sqcup$[2,3]=[1,2,3]. Any annotation contains the empty annotation and is contained in the full annotation. We write a stream containing the message m_1 before the message m_2 as $m_1 m_2$.

Although a node is not a stream message, for the sake of conciseness we may often speak about streams made up of nodes. Thus, the wordings (1) "all children of a node n are annotated in the output stream with the annotation of n from the input stream" and (2) "all opening tags of children of a node n are immediately followed in the output stream by the annotation that immediately follows the opening tag of n in the input stream" are equivalent.

B. The Query Language XPath

This article considers a clean fragment of XPath [19] defined by the following abstract EBNF:

```
query:  '/' path | path | query 'union' query | query 'except' query
path:   step ('/' step)*
step:   axis '::' nodetest ('[' pred ']' )?
pred:   pred 'and' pred | pred 'or' pred | 'not' '(' pred ')'
        | '(' pred ')' | path
```

A path is a sequence of steps and each step has an axis (i.e., a binary relation on nodes), a nodetest (i.e., wildcard '*' or a node label), and possibly a predicate (i.e., a boolean formula over paths). If a path is preceeded by '/', then it is absolute, otherwise it is relative. A query is a path, a union or a difference of paths. The semantics of a path $L_1[p_1]/\ldots/L_n[p_n]$ is the set of all nodes m_n in stream order such that there is a list of nodes $(m_0, \ldots, m_n)$, where m_0 is any node among a given set of (context) nodes, and for all $1 \leq i \leq n$ we have $(m_{i-1}, m_i) \in L_i$, and there are nodes m'_i for which $(m_i, m'_i) \in p_i$. In case of an absolute path, m_0 is by default bound to the root node.

Our XPath fragment is restricted in that it only considers forward axes. A forward axis relation α holds on two nodes n and m, if m appears after n in stream order or m equals n. XPath also defines reverse axes, which are inverses of forward axes. Note that in the case of absolute paths the restriction to forward axes does not make our XPath fragment less expressive [15], [24]. The supported forward axes are self (equality), fstChild (first child), child (child), child$^+$ (descendant), child* (descendant or self), nextSibl (next sibling), nextSibl$^+$ (next siblings), nextSibl* (next siblings or self), and foll (following).

We also define vertical, horizontal, and diagonal paths and predicates. From any node n, a vertical path selects descendants of n and possibly n itself, a horizontal path selects descendants of the parent of n, and a diagonal path selects descendants of the root. Because we consider here only forward axes, the selected nodes follow or equal n in all three cases. Syntactically, if we ignore the occurences of self, a vertical path starts with fstChild, child, or their closures, and can also contain nextSibl and its closures. A horizontal path starts with nextSibl or its closures, and can contain any axis but foll. A diagonal path can contain any axis. A vertical (horizontal, diagonal) predicate consists of vertical (horizontal, diagonal respectively) paths.

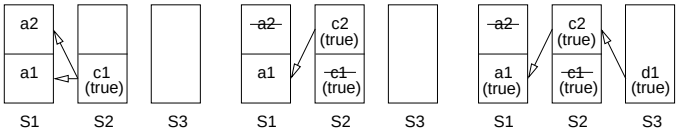
III. Overview of SPEX Processing

SPEX evaluates *one* XPath query against *one* XML stream (see `http://www.pms.ifi.lmu.de/forschung/spex/mq.html` for a SPEX extension coping with large query sets). This section discusses the processing strategy of SPEX for a query of the general form $/L_1[p_1]/\ldots/L_n[p_n]$. SPEX uses a compact data structure to encode matchings of each step L_i and one buffer for possible answer candidates. A candidate is a node matched by the last step L_n before the nodes required to evaluate all predicates p_i are encountered in the stream. For each step L_i we construct a list S_i, whose entries represent all matchings of L_i at any instant. An entry e_{i+1} representing a node x is added to S_{i+1} when the step L_{i+1} matches x from a node previously matched by L_i and represented by an entry e_i. In this case, we also have a link from e_{i+1} to e_i. Note that there can be several nodes matched by L_{i+1} from a node matched by L_i and also the same node can be matched by L_{i+1} from different nodes matched by L_i (both cases can happen if, e.g., L_{i+1} is a closure axis like descendant or next siblings). This implies there can be many-to-many links between the entries of two successive lists.

Besides adding new entries to our lists, we may also replace or remove existing entries. An entry e_i is replaced by *true* when the predicate p_i is satisfied at the matching node represented by e_i; if L_i has no predicate, then e_i is *true* by default. The instant when e_i is removed depends on the rest of the query: If $[p_i]/\ldots/L_n[p_n]$ is vertical (horizontal or diagonal), then e_i is removed when the closing tag of the matching node y it represents (the parent of the matching node y or the end of the stream respectively) is encountered in the stream. The reason why e_i is removed at that

instant is that the paths in $[p_i]/\ldots/L_n[p_n]$ can only match nodes that are descendants of y or y itself (descendants of the parent of y or of the root node respectively).

Each entry e_n of S_n represents a candidate. By following the links back from e_n to entries of S_1, we discover all dependencies of a candidate represented by e_n. A dependency of a candidate is thus a sequence $(e_n, \ldots, e_1)$ of linked entries with one entry from each list S_i. When at least one dependency of a candidate c becomes a sequence of *true* values, then c is in the answer. This is the case when each predicate p_i is satisfied at the entry e_i of that dependency ($1 \leq i \leq n$). When at least one entry of each dependency of c is removed before becoming *true*, then c is removed as well. This is the case when there is at least one predicate that is not satisfied for each of the dependencies of c.



(a) after first ⟨c⟩ (b) after second ⟨/c⟩ (c) after ⟨d⟩

Fig. 2. Partial matchings are encoded using an efficient data structure.

Example 3.1: Consider a query that selects the d-node from the input stream of Fig. 1: /child$^+$::a[child::d]/child$^+$::c/nextSibl$^+$::*. The partial matchings created for our query and stream at different processing instants are shown in Fig. 2: (a) after processing the first opening tag ⟨c⟩, (b) after processing the second closing tag ⟨/c⟩, and (c) after processing the first opening tag ⟨d⟩.

There are three lists S_1, S_2, and S_3 corresponding to our three steps. In case (a), S_1 has two entries a_1 and a_2 corresponding to the two a-nodes already read, S_2 has one entry c_1 corresponding to the first c-node, and S_3 is empty. Note that c_1 is *true* and linked with both a_1 and a_2, because the first c-node is descendant of both a-nodes. In case (b), the entry c_1 is removed (on reading the closing tag of the first c-node), because the corresponding c-node has no next sibling. Also, a_2 is removed on reading the closing tag of the second a-node, because this node has no child d-node. For clarity, we still represent the removed entries but marked as deleted. A new entry c_2 (set to *true*) is created for the second c-node and is linked to a_1, because the second c-node is a descendant of the first a-node. In case (c), a_1 is replaced by *true*, because the first predicate is satisfied on the first a-node. Also, there is a new entry d_1 (set to *true*) in S_3 corresponding to the answer candidate represented by the d-node and linked to c_2. We can now decide that this candidate is in the answer, because all its dependencies are resolved to *true*. We then output this candidate and, on reading its closing tag, we remove d_1 from S_3. After closing the first a-node, we can safely remove a_1 and c_1. □

At any instant the size of S_i is bounded in the number of matchings of L_i. Note that there can be exponentially (in n) many dependencies, although our partial matching structure represents them polynomially. The size of the candidates buffer is bounded in the stream size (though the answer size can be quadratic in the stream size). Also, the buffer is kept as small as possible by discarding candidates as soon as their predicates are evaluated.

In addition to the memory-conscious data structures for candidates and step matchings, SPEX has efficient algorithms for the structural joins represented by XPath forward axes. These algorithms are realized as basic automata with output tape, also called transducers. The main challenge in defining these transducers is to compute matchings for several input nodes at the same time and manage such matchings using only a stack. Later in this section we define our new notion of SPEX transducer, and Section V gives SPEX transducers for all XPath forward axes.

The transducers use their stacks to model the lists discussed above. Also, they use their tapes to communicate with other transducers. The communication of two transducers, say T_i and T_{i+1}, for subsequent steps L_i and L_{i+1}, is realized by making the output tape of T_i be the input tape of T_{i+1}. The communication is necessary to inform T_{i+1} of the matchings of T_i and happens along the stream as node anotations. By interconnecting the transducers for the steps constituting a query, we construct a network representing that query. Section IV defines the compilation of queries into transducer networks.

A natural choice for processing a stream with a transducer network is to let the stream flow through the network one message at a time. By default, SPEX enforces that the entire network processes a stream message before the next stream message is processed. Section VI gives a SPEX extension that departs from this rule and allows each message to be processed only by transducers that can potentially use it to create or resolve a candidate dependency.

SPEX transducers and transducer networks. Pushdown transducers are automata with pushdown store and output tape. More formally, a pushdown transducer [25] is an eight-tuple $(Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$, where

- Q is a finite set of states.
- Σ, Γ, and Δ are the input, pushdown, and respectively output alphabets.
- δ is a relation from $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$ to $2^Q \times \Gamma^* \times (\Delta \cup \{\varepsilon\})$ called the transition table, whose elements are called transition rules.
- $q_0 \in Q$ is the initial state.
- $Z_0 \in \Gamma$ is the bottom pushdown symbol.
- $F \subseteq Q$ is the subset of accepting or final states.

A deterministic pushdown transducer allows at most one transition from any of its states. In this case, the transition relation becomes a function from $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$ to $Q \times \Gamma^* \times (\Delta \cup \{\varepsilon\})$.

We use in this article a simplified class of transducers, which we call SPEX transducers.

Definition 3.2: A SPEX transducer is a single-state deterministic pushdown transducer, where the input and output alphabets are the set M of all opening and closing tags and annotations, the stack alphabet is the set A of all annotations, the bottom pushdown symbol Z_0 is the empty annotation [ ], and the transition function δ is canonically extended to the configuration-based transition function $\vdash$: $M \times A^* \rightarrow A^* \times M^*$. □

Section V gives the configuration-based transitions of SPEX transducers for all our XPath axes.

Example 3.3: Consider the SPEX transducer defined by the following transitions

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

We use the notation $[c] \mid \gamma$ to express that the stack of our transducer is split in its top $[c]$ and the rest γ. We also use ε to denote that no symbol is output.

On receiving an annotation $[c]$, the first transition pushes that symbol onto the stack and outputs nothing.

On receiving an opening tag $\langle\eta\rangle$, and with $[s]$ the top of the stack, the second transition keeps the same stack configuration and outputs $\langle\eta\rangle$ *followed* by $[s]$.

On receiving a closing tag $\langle/\eta\rangle$, and with $[s]$ the top of the stack, the third transition outputs the input symbol and pops the top annotation off the stack.

It can be checked that the annotation of each node is moved to its children. Section V shows indeed that this SPEX transducer implements the child axis. □

Transducer networks are obtained by composing transducers in sequence and parallel. If two transducers t_1 and t_2 are composed in sequence, noted $t_1 \cdot t_2$, then the output stream of t_1 is the input stream of t_2. If two transducers t_1 and t_2 are composed in parallel, noted $t_1 +\!\!+ t_2$, then they receive the same input stream.

For boolean (and, or) and set (union, except) operators we specify transducers with several input tapes. Such a transducer unifies the streams received on its input tapes by outputting each opening and closing tag from the original stream only once and after it reads that tag from all its input tapes. Additionally, according to the semantics of the implemented operator, it uses the annotations of each opening tag from all received streams to compute and output a new annotation. The transducers for boolean and set operators are given in Section V-C.

Example 3.4: Consider the following two input streams that differ only in their annotations

⟨r⟩[0] ⟨a⟩[ ] ⟨a⟩[2] ⟨c⟩[ ] ⟨/c⟩⟨/a⟩⟨c⟩[ ] ⟨/c⟩⟨d⟩[3] ⟨/d⟩⟨/a⟩⟨/r⟩
⟨r⟩[0] ⟨a⟩[1] ⟨a⟩[1] ⟨c⟩[2] ⟨/c⟩⟨/a⟩⟨c⟩[ ] ⟨/c⟩⟨d⟩[ ] ⟨/d⟩⟨/a⟩⟨/r⟩

The output of the transducer or after reading the streams is

⟨r⟩[0] ⟨a⟩[1] ⟨a⟩[1,2] ⟨c⟩[2] ⟨/c⟩⟨/a⟩⟨c⟩[ ] ⟨/c⟩⟨d⟩[3] ⟨/d⟩⟨/a⟩⟨/r⟩

Note that each opening and closing tag appears only once in the output. Also, each opening tag is annotated with the union of the annotations of that tag in the input streams. □

Finally, there are three special transducers in, out, and head.

The transducer in is the first transducer in a network and its task is to annotate the nodes from the input stream. Fig. 1 gives the output stream of transducer in, where the root node is assigned a full annotation, and the other nodes are assigned empty annotations. This annotation scheme corresponds to the evaluation of absolute paths, i.e., paths that are always evaluated from the root node. To evaluate paths from a given set of nodes, these nodes are assigned a full annotation (this corresponds then to the evaluation of relative paths).

The transducer out is the last transducer in a network and its simple task is to manage the candidates, i.e., to store, output, and discard them as soon as possible (as previously discussed in Section III). For this task, the transducer out has a random-access buffer. We skip here the specification of this transducer.

The transducer head is positioned in a network immediately after the transducer for the last query step and marks non-empty annotations with a *head* flag. Because the transducer for the last query step annotates answer candidates, the transducer head ensures that the candidates are distinguished from the other nodes in the stream.

IV. Query Compilation

SPEX compiles an XPath query into a transducer network that mirrors the structure of the query. The compilation has four distinct simple phases, which are detailed next.

1. Query Preparation Phase. We first add a new step *head* at the end of the query (or of each operand, if the query is a set operation with several operands). The semantics of head is that of a self step with a wildcard nodetest (thus, by adding it to a query we do not change the query semantics). Second, we annotate each predicate with an identifier and with the type of the predicate and of the paths following that predicate in the query. Recall from Section II that XPath paths and predicates can be vertical (v for short), horizontal (h), or diagonal (d).

Example 4.1: The predicate [child::a] in the query /child::b[child::a]/nextSibl$^+$::c becomes $[\text{child::a}]_1^h$, because the predicate is vertical and the path following it is horizontal.□

$$
\begin{aligned}
[\![p_1/p_2]\!] &= [\![p_1]\!] \cdot [\![p_2]\!] \\
[\![p_1[p_2]_n^x]\!] &= [\![p_1]\!] \cdot [\![[p_2]_n^x]\!] \\
[\![[p]_n^x]\!] &= \overrightarrow{scope}_n^x \cdot ([\![p]\!]) \cdot \overleftarrow{scope}_n^x \\
[\![p_1 \ \mathsf{op} \ p_2]\!] &= ([\![p_1]\!] +\!\!+ [\![p_2]\!]) \cdot \mathsf{op} \\
[\![\mathsf{not}(p)]\!] &= [\![p]\!] \cdot \mathsf{not} \\
[\![(p)]\!] &= [\![p]\!] \\
[\![\alpha::\eta]\!] &= \alpha \cdot \eta \\
[\![\mathit{head}]\!] &= \boxed{\mathit{head}}
\end{aligned}
$$

Fig. 3. Query compilation phase.

2. Query compilation phase. The compilation is given in Fig. 3 by the function $[\![\ ]\!]$ defined using pattern matching on the structure of XPath queries. The operator op is one of union, except, and, or. For each predicate with identifier n and type x we create a block ($\overrightarrow{scope}_n^x$, $\overleftarrow{scope}_n^x$) in the network. The XPath operators '/' and '::' are translated into sequential compositions, and for each operator op we compose in parallel the networks for its operands. Note that we overload the names of operators, axes, and nodetests to also denote transducers. Also, while the operators are infix in queries, their corresponding transducers are postfix in networks.

$$
\begin{aligned}
(X \cdot \mathsf{op'} +\!\!+ Y) \cdot \mathsf{op'} &\rightarrow (X +\!\!+ Y) \cdot \mathsf{op'} && (1)\\
X \cdot Y +\!\!+ X \cdot Z &\rightarrow X \cdot (Y +\!\!+ Z) && (2)\\
X +\!\!+ X \cdot Z &\rightarrow X \cdot Z && (3)\\
\overrightarrow{scope}_n^x \cdot X \cdot \overleftarrow{scope}_n^x \cdot Y &\rightarrow \overrightarrow{scope}_n^x \cdot (X +\!\!+ Y) \cdot \mathsf{and} \cdot \overleftarrow{scope}_n^x && (4)
\end{aligned}
$$

Fig. 4. Network rewriting phase.

3. Network rewriting phase. The transducer networks produced in the compilation phase can be further minimized using the term rewriting system defined in Fig. 4, where the variables X, Y, and Z stand for arbitrary networks. Although not shown here, it can be checked that the system is terminating and confluent.

Rule (1) eliminates redundant commutative and associative operators (op' is and, or, or union). For example,

$$
\begin{aligned}
&((\mathsf{child} +\!\!+ \mathsf{child}^+) \cdot \mathsf{and} +\!\!+ \mathsf{nextSibl}) \cdot \mathsf{and} \rightarrow \\
&(\mathsf{child} +\!\!+ \mathsf{child}^+ +\!\!+ \mathsf{nextSibl}) \cdot \mathsf{and}
\end{aligned}
$$

Rules (2) and (3) factor out common prefixes of subnetworks

composed in parallel. For example,

$$\mathsf{child}\cdot\mathsf{a}\cdot\mathsf{nextSibl}^+ \mathbin{+\!\!+} \mathsf{child}\cdot\mathsf{b}\cdot\mathsf{child}^+ \rightarrow$$
$$\mathsf{child}\cdot(\mathsf{a}\cdot\mathsf{nextSibl}^+ \mathbin{+\!\!+} \mathsf{b}\cdot\mathsf{child}^+)$$

Rule (4) composes in parallel the subnetworks for predicates and for their following subqueries. For example,

$$\overrightarrow{scope}_1^{\,v}\cdot\mathsf{child}\cdot\mathsf{a}\cdot\overleftarrow{scope}_1^{\,v}\cdot\boxed{\mathit{head}}\rightarrow$$
$$\overrightarrow{scope}_1^{\,v}\cdot(\mathsf{child}\cdot\mathsf{a}\mathbin{+\!\!+}\boxed{\mathit{head}})\cdot\mathsf{and}\cdot\overleftarrow{scope}_1^{\,v}$$

4. Network fixup phase. We finally compose in sequence the transducer in, the outcome of the previous rewriting phase, and the transducer out.

Example 4.2: Consider a query that selects all d children of a-nodes that are children of the root and have descendants b and children c: /child::a[child$^+$::b and child::c]/child::d.

After the preparation phase, the query becomes

/child::a[child$^+$::b and child::c]$_1^v$/child::d/*head*

The compilation phase yields

$$\mathsf{child}\cdot\mathsf{a}\cdot\overrightarrow{scope}_1^{\,v}\cdot(\mathsf{child}^+\cdot\mathsf{b}\mathbin{+\!\!+}\mathsf{child}\cdot\mathsf{c})\cdot\mathsf{and}\cdot\overleftarrow{scope}_1^{\,v}\cdot\mathsf{child}\cdot\mathsf{d}\cdot\boxed{\mathit{head}}.$$

Finally, the rewriting and fixup phases yield $\mathsf{in}\cdot\mathsf{child}\cdot\mathsf{a}\cdot\overrightarrow{scope}_1^{\,v}\cdot(\mathsf{child}^+\cdot\mathsf{b}\mathbin{+\!\!+}\mathsf{child}\cdot(\mathsf{c}\mathbin{+\!\!+}\mathsf{d}\cdot\boxed{\mathit{head}}))\cdot\mathsf{and}\cdot\overleftarrow{scope}_1^{\,v}\cdot\mathsf{out}$ □

V. Evaluation with Transducer Networks

This section defines SPEX transducers that implement the XPath forward axes and nodetests. Sequential compositions of SPEX transducers implement then queries without predicates. For queries with predicates and set operators, we give additional transducers for handling predicates, as well as for boolean and set operators.

A. SPEX Transducers for Forward Axes and Nodetests

Given a tree T and a set of context nodes in T, the evaluation of a forward axis α yields the set of all nodes in T that stand in relation α with at least one context node. Provided the context nodes are marked with non-empty annotations in the input stream conveying T, the transducer implementing α outputs a stream that also conveys T and where the nodes that stand in relation α with some context nodes are assigned the annotations of their corresponding context nodes. Note that in general there can be several nodes that stand in relation α with the same context node (for any axis relation but self and nextSibl), and even with several context nodes (for closure relations like child$^+$). It is crucial for the efficiency of our approach that a SPEX transducer for a forward axis α can annotate correctly in *one pass* over the input stream the nodes in T while using only a stack to keep track of the depth of the nodes in the stream and to store annotations read from the input stream.

Configuration-based transitions defining SPEX transducers for forward axes are given next. Initially, an empty annotation [ ] is pushed onto the stack of each transducer. These transducers only differ in their first transitions, which are compactions of simpler transitions that do only one stack operation.

The transducer child moves the annotations of nodes to their children. The transitions of this transducer read as follows: (1) if an annotation $[c]$ is received, then $[c]$ is pushed onto the stack and nothing is output; (2) if an opening tag $\langle\eta\rangle$ is received, then it is output followed by the top of the stack; (3) if a closing tag is received, then it is output and the stack is popped.

1. $([c]\ ,\ \ \gamma) \vdash ([c]\mid\gamma,\ \ \varepsilon)$
2. $(\langle\eta\rangle\ ,[s]\mid\gamma)\vdash([s]\mid\gamma,\langle\eta\rangle[s])$
3. $(\langle/\eta\rangle,[s]\mid\gamma)\vdash(\ \ \gamma,\ \langle/\eta\rangle)$

Recall that the annotation of a node n follows its opening tag. When receiving a node n annotated with $[c]$, $[c]$ is pushed onto the stack. The following two cases can then appear:

(1) the closing tag of n is received, and $[c]$ is popped off the stack. This corresponds to the case when there are no other children of n left in the input stream.
(2) the opening tag of a child node m of n is received, and it is output followed by $[c]$. Thus, the node m is annotated correctly with $[c]$, which was the annotation of n.

In the second case, a new annotation, say $[c']$, is received afterwards, pushed onto the stack, and used to annotate children p of m. Only when the closing tag of p is received, $[c']$ is popped and $[c]$ becomes again the top of the stack. At this time, siblings of m can be received and annotated with $[c]$ (the above cases 2), or the closing tag of n is received (the above case 1).

The transducer fstChild moves the annotations of nodes to their first children. This transducer is a simplification of the child-transducer, by restricting a stored annotation $[s]$ of a node n to mark at most one node. This node is necessarily the first child of n, as ensured by the stream's sequence. This restriction can be realized by replacing $[s]$ with the empty annotation as soon as a child of n and its annotation, say $[c]$, is received. Below, we give the first transition modified accordingly. The other transitions are as for the transducer child.

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

The transducer nextSibl moves the annotations of nodes to their immediate next sibling, if any. The transitions of this transducer are the same as for the transducer child, except for the first one given below. In the first transition, the top of the stack $[s]$ is replaced with the received annotation $[c]$ of a node n and pushes an empty annotation [ ] onto the stack. The annotation [ ] is then used to annotate children of n. When the closing tag of n is received, the annotation [ ] is popped and its next sibling node m can be annotated with $[c]$. The other next siblings can not be annotated with $[c]$, because $[c]$ is replaced by the annotation of m, say $[c']$, and now the immediate next sibling of m can be annotated with $[c']$.

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

The transducer child$^+$ moves the annotations of nodes to their descendants. The transitions of this transducer are the same as for the child-transducer, except for the first one given below. In the first transition, this transducer pushes onto the stack the received annotation $[c]$ *together* with the top annotation $[s]$: $[c]\sqcup[s]$. The difference to the transducer child is that also the annotations $[s]$ of the ancestors n_a of n are used to annotate children m of n, for the nodes m are also descendants of the nodes n_a.

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

When receiving a node n annotated with $[c]$, $[c]$ is pushed onto the stack together with the current top $[s]$: $[c]\sqcup[s]$. The following two cases can then appear:

(1) the closing tag of n is received, and $[c]\sqcup[s]$ is popped off the stack. This corresponds to the case when there are no other descendants of n left in the incoming stream.
(2) the opening tag of a child m of n is received, and it is output followed by $[c]\sqcup[s]$. Thus, the children of n, which are also descendants of n, are annotated correctly.

input	$\langle a\rangle$	[1]	$\langle a\rangle$	[2]	$\langle b\rangle$	[3]	$\langle/b\rangle$	$\langle/a\rangle$	$\langle b\rangle$	[ ]	$\langle/b\rangle$	$\langle/a\rangle$
child::b	$\langle a\rangle$	[ ]	$\langle a\rangle$	[ ]	$\langle b\rangle$	[2]	$\langle/b\rangle$	$\langle/a\rangle$	$\langle b\rangle$	[1]	$\langle/b\rangle$	$\langle/a\rangle$
child$^+$::b	$\langle a\rangle$	[ ]	$\langle a\rangle$	[ ]	$\langle b\rangle$	[1,2]	$\langle/b\rangle$	$\langle/a\rangle$	$\langle b\rangle$	[1]	$\langle/b\rangle$	$\langle/a\rangle$
nextSibl$^+$::b	$\langle a\rangle$	[ ]	$\langle a\rangle$	[ ]	$\langle b\rangle$	[ ]	$\langle/b\rangle$	$\langle/a\rangle$	$\langle b\rangle$	[2]	$\langle/b\rangle$	$\langle/a\rangle$
foll::b	$\langle a\rangle$	[ ]	$\langle a\rangle$	[ ]	$\langle b\rangle$	[ ]	$\langle/b\rangle$	$\langle/a\rangle$	$\langle b\rangle$	[2,3]	$\langle/b\rangle$	$\langle/a\rangle$

Fig. 5. Processing example with SPEX transducers.

In the second case, a new annotation, say $[c']$, is received afterwards, the annotation $[c']\sqcup[c]\sqcup[s]$ is pushed onto the stack and used to annotate children p of m. Thus, the annotation $[c]$ is also used to annotate children p of m (n''), hence descendants of n. Only when the closing tag of p is received, $[c']\sqcup[c]\sqcup[s]$ is popped and $[c]\sqcup[s]$ becomes again the top of the stack. At this time, siblings of m can be received and annotated with $[c]\sqcup[s]$ (the above case 2), or the closing tag of n is received (the above case 1).

The transducer nextSibl$^+$ moves the annotations of nodes to their next siblings. The transitions of this transducer are the same as for the child-transducer, except for the first one given below. In the first transition, this transducer adds to the top of the stack $[s]$ the received annotation $[c]$ of the source node n and pushes an empty annotation [ ]. The annotation [ ] is then used to annotate children of n. When the closing tag of n is received, the annotation [ ] is popped and its next sibling nodes m can be annotated with the top annotation $[c]$. Because the old top of the stack $[s]$ is kept together with the newly received annotation $[c]$, the annotations of preceding siblings of n are also used to annotate the following siblings of n.

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

The transducer child* moves the annotations of each node n to its descendants and to the node n itself. This transducer is defined below similar to the transducer child$^+$, with the difference that a node n keeps its own annotation $[c]$ together with the annotations $[s]$ of its ancestors .

$$\begin{array}{l}1.\ ([c]\ ,\ [s]\mid\gamma)\vdash([c]\sqcup[s]\mid[s]\mid\gamma,\ [c]\sqcup[s])\\ 2.\ (\langle\eta\rangle\ ,\ \ \ \gamma)\vdash(\ \ \ \ \ \ \ \ \ \ \gamma,\ \ \ \langle\eta\rangle)\\ 3.\ (\langle/\eta\rangle,\ [s]\mid\gamma)\vdash(\ \ \ \ \ \ \ \ \ \ \gamma,\ \ \ \langle/\eta\rangle)\end{array}$$

The transducer nextSibl* moves the annotations of each node n to its next siblings and to the node n itself.

$$\begin{array}{l}1.\ ([c]\ ,\ [s]\mid\gamma)\vdash([\,]\mid[c]\sqcup[s]\mid\gamma,\ [c]\sqcup[s])\\ 2.\ (\langle\eta\rangle\ ,\ \ \ \gamma)\vdash(\ \ \ \ \ \ \ \ \ \ \gamma,\ \ \ \langle\eta\rangle)\\ 3.\ (\langle/\eta\rangle,\ [s]\mid\gamma)\vdash(\ \ \ \ \ \ \ \ \ \ \gamma,\ \ \ \langle/\eta\rangle)\end{array}$$

A nodetest η is a unary relation. For a given set of context nodes, it returns a subset of this set consisting of the nodes with that nodetest. This means that the initial and returned sets are the same in the case of a wildcard nodetest. We therefore create no transducer for a wildcard nodetest.

A transducer for a nodetest η replaces the annotations of nodes without that nodetest by the empty annotation. The transitions of this transducer are given next. For simplification, each transition can consider two input symbols at a time. The nodetest $\neg\eta$ stands for any nodetest but η from our finite set of nodetests.

$$\begin{array}{l}1.\ (\langle\eta\rangle[c]\ \ )\vdash(\langle\eta\rangle[c])\\ 2.\ (\langle\neg\eta\rangle[c])\vdash(\ \langle\eta\rangle[\,]\ )\\ 3.\ (\langle/\eta\rangle\ \ \ \ )\vdash(\ \ \langle/\eta\rangle)\\ 4.\ (\langle/\neg\eta\rangle\ \ )\vdash(\langle/\neg\eta\rangle)\end{array}$$

Variations of Transducers for Forward Axes. We can summarize the transitions of the previously defined transducers as follows ($[c]$ is the annotation of n currently read and $[s]$ is the current top of the stack):

1. $[c]$ is output as soon as it is read. Then, $[c]$ is used to mark also n.
2. $[c]$ is pushed onto the stack. Then, $[c]$ is used to mark also the children of n.
3. $[c]$ is pushed one level below the top. Then, $[c]$ is used to mark also the next sibling of n.
4. $[s]$ is onto the stack. Then, $[s]$ is used to mark also the descendants of n.
5. $[s]$ is pushed one level below the top. Then, $[s]$ is used to mark also the next siblings of n.

By mixing the above behaviors 1 to 5, one can get a transducer for any axis. For example, combining behaviors 1 and any other ensures the reflexivity of the axis. Combining behaviors 4 and 2, or 5 and 3, ensures the transitivity of the axis. Combining 1 and 2 and 4, or 1 and 3 and 5, ensures both the transitivity and reflexivity of the axis.

There are, of course, other possible combinations. For example, the combination of 2 to 5 gives the implementation of the complex relation child$^+$-or-nextSibl$^+$ = child$^+$ $\cup$ nextSibl$^+$. These combinations are reflected in the following changed transition:

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

We next define the transducer foll. In the first transition, it replaces the old top annotation $[s]$ with the new annotation $[c]$ and then pushes also the old top $[s]$. Because the nodes following a node n are all nodes reachable in the further stream after closing n, the annotation $[c]$ becomes part of the top of the stack and used to annotate incoming nodes as soon as the node n is closed (transition 3). In contrast to the transducers defined previously, once an annotation becomes part of the stack, it remains there, because the following sibling nodes of the ancestor nodes of n follow also n.

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

Although pushdown transducers are not closed under composition, the composition of pushdown and finite transducers is possible and even beneficial. In this sense, one can create transducers implementing composition of axes and nodetests. We give below the transitions of the transducer child::a for the composition of the child axis and the a nodetest defining the step child::a.

$$\begin{array}{l}1.\ ([c]\ \ \ ,\ \ \ \ \gamma)\vdash([c]\mid\gamma,\ \ \ \ \ \varepsilon)\\ 2.\ (\langle\mathsf{a}\rangle\ \ \ ,\ [s]\mid\gamma)\vdash([s]\mid\gamma,\ \ \langle\mathsf{a}\rangle[s])\\ 3.\ (\langle\neg\mathsf{a}\rangle\ ,\ \ \ \ \gamma)\vdash(\ \ \ \ \gamma,\ \langle\neg\mathsf{a}\rangle[\,])\\ 4.\ (\langle/\mathsf{a}\rangle\ \ ,\ [s]\mid\gamma)\vdash(\ \ \ \ \gamma,\ \ \ \langle/\mathsf{a}\rangle)\\ 5.\ (\langle/\neg\mathsf{a}\rangle,\ [s]\mid\gamma)\vdash(\ \ \ \ \gamma,\ \ \langle/\neg\mathsf{a}\rangle)\end{array}$$

Example 5.1: Figure 5 gives the output streams of the SPEX transducers child$^+$::b, nextSibl$^+$::b, and foll::b after processing an input stream. We explain how the transducer child::b processes incrementally that input stream.

Recall that the stack is initialized with an empty annotation [ ]. The stack configuration changes only on receiving annotations and closing tags. On receiving opening tags matching its nodetest,

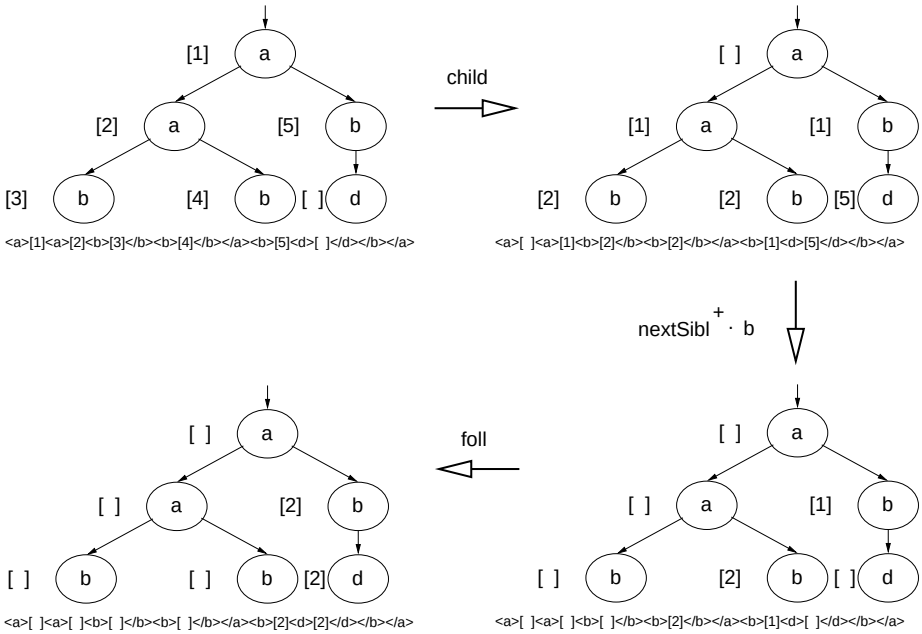


Fig. 6. Evaluating the query child::*/nextSibl$^+$::b/foll::* with network child · nextSibl$^+$ · b · foll.

the transducer outputs that opening tag followed by the top of its stack.

- ⟨a⟩ is output followed by its top annotation [ ]. Thus, the first a-node does not have in the input stream a parent with a non-empty annotation. The stack configuration remains [ ].
- [1] is pushed onto the stack, This way, it is instructed to mark all b-children of the first a-node with [1]. The stack configuration becomes [1]|[ ] (the top is at the left).
- ⟨a⟩ is output followed by [ ]. Although the top annotation is [1], this output is correct, because the received node does not have a b-nodetest. The stack configuration remains [1]|[ ].
- [2] is pushed onto the stack. This way, it is instructed to mark all b-children of the second a-node with [2]. The stack configuration becomes [2]|[1]|[ ].
- ⟨b⟩ is output followed by the top annotation [2]. This output is correct, because the received node does have a b-nodetest and is a child of the second a-node. The stack configuration remains [2]|[1]|[ ].
- [3] is pushed onto the stack. This way, it is instructed to mark all b-children of the first b-node with [3]. The stack configuration becomes [3]|[2]|[1]|[ ].
- ⟨/b⟩ It pops the top [3] off the stack, meaning that there are no children of the first b-node left in the stream. This is correct, because the first b-node does not have children at all. The stack configuration becomes [2]|[1]|[ ].
- ⟨/a⟩ It pops the top [2] off the stack, meaning that there are no children of the second a-node left in the stream. The stack configuration becomes [1]|[ ].
- ⟨b⟩ It outputs the tag, followed by the top annotation [1]. This output is correct, because the received node does have a b-nodetest and is a child of the first a-node. The stack configuration remains [1]|[ ].
- [ ] is pushed onto the stack. This way, it is instructed to mark all b-children of the second b-node with [ ]. Because the other children are also marked with [ ], we can conclude that the transducer will mark all children of the second b-node with [ ]. The stack configuration becomes [ ]|[1]|[ ].
- ⟨/b⟩ It pops the top [ ] off the stack, meaning that there are no children of the second b-node left in the stream. The stack configuration becomes [1]|[ ].
- ⟨/a⟩ It pops the top [1] off the stack, meaning that there are no children of the first a-node left in the stream. The stack configuration becomes [ ] and the processing is finished. □

B. Transducer Networks for Location Paths without Predicates

SPEX compiles a query without predicates into a network representing a sequence of transducers for the constituent location steps. The network processes then the input stream and the nodes annotated in the output stream represent the answer to that query.

Example 5.2: Consider the query child::*/nextSibl$^+$::b/foll::* that selects from any context node all nodes that follow the b-labeled siblings of its children. SPEX compiles this query into the network in · child · nextSibl$^+$ · b · foll · out. Consider that in annotates the input stream as given in the top-left tree of Fig. 6. The stream of the bottom-left tree of Fig. 6 represents the output of the transducer foll, where only the last two nodes have the non-empty annotations. This means that only these two nodes stand in relation child::*/nextSibl$^+$::b/foll::* with nodes from the input stream. By inspecting their annotations, we conclude that both of them are selected from the second a-node. The transducer out outputs these two nodes in stream order. Fig. 6 also shows as annotated trees and streams the intermediary results of the transducers child, nextSibl$^+$::b, and foll, albeit they are not materialized during processing. □

C. Handling Set Operators

The transducers union and except have several input tapes. Their common task is to unify the streams received on the input tapes by outputting each opening and closing tag from the original stream only once and when it is read from all input tapes. A non-empty annotation is output if it appears in at least one input

in	⟨⟩	[0]	⟨a⟩	[ ]	⟨a⟩	[ ]	⟨c⟩	[ ]	⟨/c⟩	⟨/a⟩	⟨c⟩	[ ]	⟨/c⟩	⟨d⟩	[ ]	⟨/d⟩	⟨/a⟩	⟨/⟩
union	⟨⟩	[ ]	⟨a⟩	[0]	⟨a⟩	[0]	⟨c⟩	[0]	⟨/c⟩	⟨/a⟩	⟨c⟩	[0]	⟨/c⟩	⟨d⟩	[ ]	⟨/d⟩	⟨/a⟩	⟨/⟩
child · child	⟨⟩	[ ]	⟨a⟩	[ ]	⟨a⟩	[0]	⟨c⟩	[ ]	⟨/c⟩	⟨/a⟩	⟨c⟩	[0]	⟨/c⟩	⟨d⟩	[0]	⟨/d⟩	⟨/a⟩	⟨/⟩
except	⟨⟩	[ ]	⟨a⟩	[0]	⟨a⟩	[ ]	⟨c⟩	[0]	⟨/c⟩	⟨/a⟩	⟨c⟩	[ ]	⟨/c⟩	⟨d⟩	[ ]	⟨/d⟩	⟨/a⟩	⟨/⟩

Fig. 7. Processing example with transducers for set operators.

stream (for union), or in the first stream and not in the others (for except); otherwise, it is replaced by the empty annotation.

Example 5.3: Consider a query that selects all nodes labeled a or c without the second-level nodes in the stream: /child$^+$::a union /child$^+$::c except /child::*/child::*. For the stream given in Fig. 1, this query selects the first a-node and the first c-node in stream order. The corresponding network is

$$\mathsf{in}\cdot(\mathsf{child}^+\cdot(\mathsf{a} +\!\!+ \mathsf{c})\cdot\mathsf{union} +\!\!+ \mathsf{child}\cdot\mathsf{child})\cdot\mathsf{except}\cdot\boxed{\mathsf{head}}\cdot\mathsf{out}.$$

The output streams of some transducers from the network are given in Fig. 7.

The entire network processes each stream message at a time. The root node is marked by the transducer in with the full annotation [0]. The opening tag of the root node reaches the transducers child$^+$ and child, which record its annotation to later mark the descendants, respectively children of the root. The second node has an empty annotation when it reaches the transducers child$^+$ and child. Both these transducers match the node and annotate it with [0]. Among the remaining transducers, only the nodetest transducer a matches the node and sends it further with the same annotation. The transducer union receives then this node on both its input tapes, one time with the empty annotation (from nodetest transducer c) and one time with the full annotation (from nodetest transducer a). Similar to union, the transducer except receives this node on both its input tapes, one time with full annotation (from the transducer union) and one time with the empty annotation. The transducer head marks then the node as answer candidate and the transducer out decides it is an answer node and starts outputting it. Until the second node labeled c is read, no other candidate is encountered. When this node reaches the transducer out, it becomes an answer node. However, because this node is a descendant of the first answer node, it is buffered until the first answer node is completely output. Then, it is also output. □

D. Transducer Networks for Queries with Predicates

As specified in Section IV, a predicate $[X]_n^x$ is compiled into a network $\overrightarrow{scope}_n^x \cdot [\![X]\!] \cdot \overleftarrow{scope}_n^x$. The purpose of $\overrightarrow{scope}_n^x$ is to reannotate with fresh annotations the nodes that have non-empty annotations in the stream and to create mappings between the received annotations and the fresh ones. These fresh annotations are used by the subnetwork $[\![X]\!]$ to evaluate the logic of the corresponding predicate. Finally, $\overleftarrow{scope}_n^x$ uses the mappings created by $\overrightarrow{scope}_n^x$ to map back the received annotations to the original ones. This reannotation allows SPEX to evaluate predicates in a modular way.

Annotation Scopes. Each annotation created by $\overrightarrow{scope}_n^x$ has a scope or lifetime that depends on x. The scope of an annotation starts with the opening tag of the node, say n, having that annotation and it ends with the first closing tag after the last possible node that can be matched by the paths of X from the context node n. This is the closing tag of n for a vertical scope, the closing tag of the parent of n for a horizontal scope, and the end of the stream for a diagonal scope. Annotations and the answer candidates depending on them can be discarded as soon as their scope is exhausted. The implication of discarding the annotations as soon as possible is twofold. First, SPEX only buffers at any instant answer candidates with unresolved dependencies; candidates with resolved dependencies are discarded from the buffer as soon as possible. Second, at any instant the amount of annotations alive for $\overrightarrow{scope}_n^x$ is bounded in (a) the maximum tree depth for a vertical scope ($x = v$), (b) the sum of the maximum tree depth and breadth for a horizontal scope ($x = h$), and (c) the number of nodes in the tree for a diagonal scope ($x = d$).

Example 5.4: Fig. 8 shows how $\overrightarrow{scope}_1^v$ and $\overrightarrow{scope}_1^h$ reannotate a given stream. □

Scope Transducers. We give next the definition of $\overrightarrow{scope}_n^v$. The stack of the transducer is used as a counter that increases with every received annotation and decreases with every received closing tag. The counter is initialized with 1. For a received annotation, this transducer creates a fresh annotation representing a singleton list containing the current value of the counter. Moreover, the transducer inserts in the stream a mapping between the received annotation, say $[c]$, and the fresh annotation, say $[s]$: $[c] \xrightarrow{n} [s]$. At the end of its lifetime, the annotation $[s]$ is discarded by inserting in the stream a mapping $[\,] \xleftarrow{n} [s]$. Two of the transition rules of $\overrightarrow{scope}_n^v$ are given below. The transitions for the other message types simply copy the messages from the input to the output stream and are skipped here. As an optimization (not defined by our simplified transitions below), we may create fresh annotations if the received annotation is non-empty.

1. $([c]\ ,\ s) \vdash (s+1,\quad [s]([c] \xrightarrow{n} [s]))$
2. $(\langle/\eta\rangle,\ s) \vdash (s-1,\ \langle/\eta\rangle([\,] \xleftarrow{n} [s-1]))$

The transducer $\overleftarrow{scope}_n^x$ replaces each non-empty annotation $[c]$ encountered in the stream with the union $[s]$ of all annotations that are mapped to subsets of $[c]$. It also creates the mapping $[s] \xleftarrow{n} [c]$.

Boolean Transducers. Like the transducers for set operators, the transducers and and or have several input tapes. Their common task is to unify the streams received on the input tapes by outputting each opening and closing tag from the original stream only once and when it is read from all input tapes. A non-empty annotation is output only if it appears in at least one input stream (for or), or it already appeared in all input streams (for and).

The transducer or behaves precisely as the transducer union. The definition of and is given below as a modified SPEX transducer without stack, but with an array, whose size is given by the number k of its input tapes. The transitions for messages of other types simply copy such messages to the output. Below, X stands for $([s_1], \ldots, [s_k])$ and Y for $([s_1] \sqcup [c_1], \ldots, [s_k] \sqcup [c_k])$.

1. $(([c_1], \ldots, [c_k]),\ X) \vdash (Y,\ \prod_{i=1}^{k} ([s_i] \sqcup [c_i]))$
2. $((\langle\eta\rangle,\ldots,\langle\eta\rangle\ ,\ X) \vdash (X,\quad \langle\eta\rangle)$
3. $((\langle/\eta\rangle,\ldots,\langle/\eta\rangle)\ ,\ X) \vdash (X,\quad \langle/\eta\rangle)$

The transducer not uses the mappings created by $\overrightarrow{scope}$ to invalidate annotations. An annotation is forwarded by the trans-

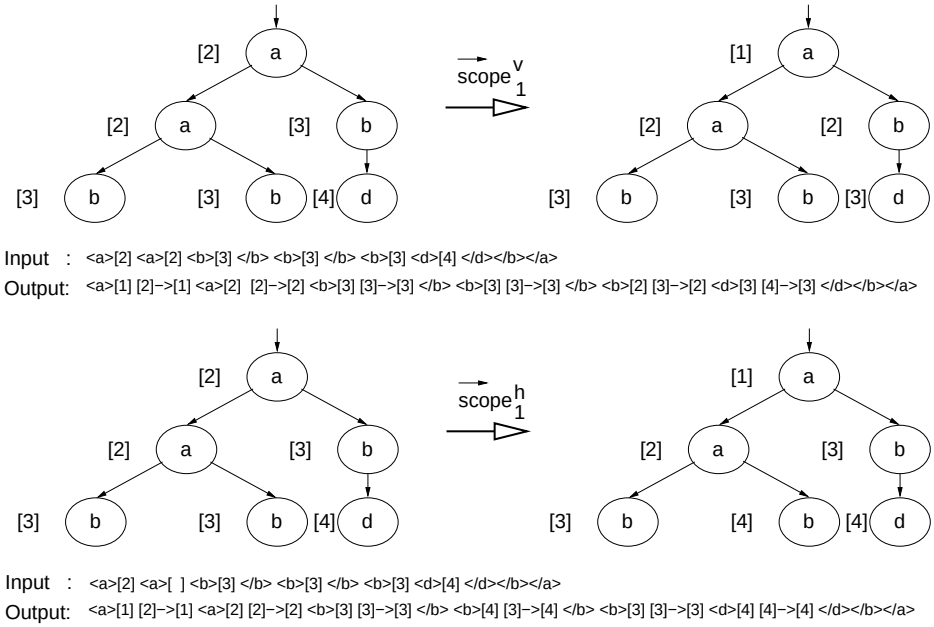


Fig. 8. Examples with vertical and horizontal scopes.

ducer not immediately before it gets invalidated, and only if the transducer not does not receive it during its lifetime.

Example 5.5: Consider the horizontal predicate [(child$^+$::a and child$^+$::c) or nextSibl::b]. From a given set of context nodes, this predicate selects only those that have descendants labeled a and c, or have an immediate sibling labeled b. For the tree with annotated nodes given in Fig. 9, the first b-node and the first a-node in stream order satisfy the predicate (the predicate is satisfied for the a-node even twice because it has descendants a and c and also an immediate sibling b).

One possible network corresponding to this predicate is $\overrightarrow{scope}_1^{\,h} \cdot ((\mathsf{child}^+ \cdot \mathsf{a} \mathbin{+\!\!+} \mathsf{child}^+ \cdot \mathsf{c}) \cdot \mathsf{and} \mathbin{+\!\!+} \mathsf{nextSibl} \cdot \mathsf{b}) \cdot \mathsf{or} \cdot \overleftarrow{scope}_1^{\,h}$.

Fig. 9 shows the input and output streams of $\overleftarrow{scope}_1^{\,h}$ and $\overrightarrow{scope}_1^{\,h}$. The output stream of the network contains the annotations of those input nodes that satisfy the predicate. Also, these annotations appear in the output stream as soon as possible. This means that before encountering the opening tag of c, it is not known whether any of the context nodes satisfy the predicate. □

VI. Reducing the Stream Traffic in Transducer Networks

The transducers introduced in Section V receive, process, and forward all nodes from the input stream, although this is by far not necessary. Ideally, for a given query, a node from the input stream should only be processed if it is critical for the correct evaluation of that query. Using SPEX terminology, these are the nodes that create or resolve candidate dependencies. We next introduce so-called filter transducers to reduce the number of nodes communicated between transducers in networks.

Vertical, Horizontal, and Diagonal Filters. We exemplify filters on a (DBLP-like) stream containing articles possibly followed at the very end of the stream by books. Consider the query /child$^+$::book[X]/child::authors asking for authors of books with given properties (X stands for the XPath encoding of these properties). The SPEX network for this query is (we assume the type of X is x)

$$\mathsf{in} \cdot \mathsf{child}^+ \cdot \mathsf{book} \cdot \overrightarrow{scope}_1^{\,x} \cdot (\mathsf{child} \cdot \mathsf{authors} \cdot \boxed{\mathit{head}} \mathbin{+\!\!+} [\![X]\!]) \cdot \mathsf{and} \cdot \overleftarrow{scope}_1^{\,x} \cdot \mathsf{out}$$

When the transducer book encounters a book-node, then the node is sent further to the successor transducers, with an additional non-empty annotation signaling a match. In the case of nodes with different labels and preceeding all book-nodes in the stream, there is no need to send them further, as they are not critical to the query (the answers to our query against the streams with or without these nodes respectively are the same). We can reduce the stream traffic between transducers in (at least) two ways.

(A). Because all transducers following the transducer book in the network are always interested in nodes following book-nodes, the query evaluation is not altered, if the transducer book sends further only the nodes following the opening tag of the first node book, and the other transducers do the same for the nodes they are instructed to find relative to nodes found by their previous transducers.

(B). Assume the transducers positioned after the transducer book in the network are only interested in descendants of book-nodes. Then, the transducer book can safely send further only the stream fragments corresponding to book-nodes.

Both aforementioned stream traffic reductions can be easily supported by SPEX extended at compile-time with so-called filter pushdown transducers. For example, in case (B), a *vertical* filter (vfilter for short) placed immediately after the transducer book, sends further only stream fragments corresponding to book-nodes. The network with vfilter is

$$\mathsf{in} \cdot \mathsf{child}^+ \cdot \mathsf{book} \cdot \mathsf{vfilter} \cdot \overrightarrow{scope}_1^{\,v} \cdot (\mathsf{child} \cdot \mathsf{authors} \cdot \boxed{\mathit{head}} \mathbin{+\!\!+} [\![X]\!]) \cdot \mathsf{and} \cdot \overleftarrow{scope}_1^{\,v} \cdot \mathsf{out}$$

In case (A), this filter is a *diagonal* filter (dfilter for short) and sends further only stream fragments starting with an opening tag book. Clearly, vertical filters make more sense for our case (B),

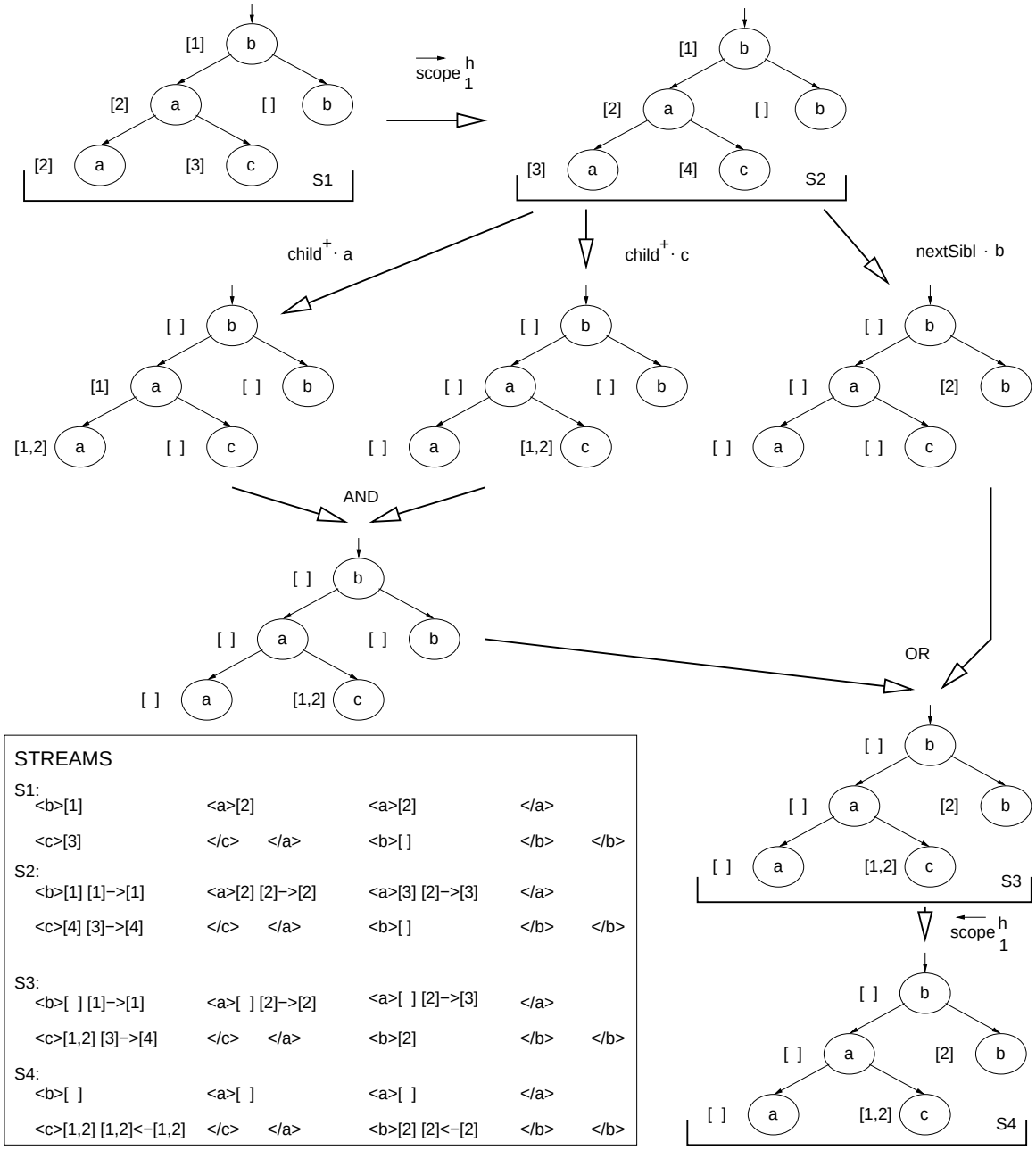


Fig. 9. Processing with network $\overrightarrow{scope}_1^h \cdot ((\text{child}^+ \cdot \text{a} \mathbin{+\!\!+} \text{child}^+ \cdot \text{c}) \cdot \text{and} \mathbin{+\!\!+} \text{nextSibl} \cdot \text{b}) \cdot \text{or} \cdot \overleftarrow{scope}_1^h$.

because they always forward smaller (or equally large) stream fragments than diagonal filters. In general, diagonal filters are not always superseded by vertical filters. It is enough to consider that the subquery X refers to nodes *following* book-nodes. The subnetwork $[\![X]\!]$ must then check for such nodes until the end of the stream, and not only inside book-nodes.

If the subquery X refers to following siblings of book-nodes, then it is sufficient to forward only all following siblings and the descendants of book-nodes. This can be achieved by placing a *horizontal* filter (hfilter for short) after the transducer book.

Remark 6.1: As illustrated above, the type of filters (vertical, horizontal, or diagonal) can be inferred from the query at compile-time, as it is the case of the transducers $\overrightarrow{scope}$. A filter of type x is placed above a subnetwork $[\![X]\!]$, if the subquery X has only paths of type x. □

Efficiency of Filters. The improvement achieved by filters depends tremendously on the selectivity of the query evaluated by the network. In the previous example, the selectivity is rather high, because the transducer book, positioned near the top of the network, finds book-nodes only at the end of a possibly large stream. In such cases, the usage of filters is fully rewarding and the evaluation resumes to mere parsing. However, in cases where the query is not selective, the additional effort to run the filters can be reflected in worse evaluation time. Section VIII shows that the average time for the evaluation of hundreds of generated queries is improved by filters up to several times.

Implementation of Filters. Fig. 10 gives the configuration-based transition function of the diagonal and vertical filters. For more compact definitions, we let _ stand for any annotation and may read two input symbols at once. Note that this relaxation does not

1. $(\langle\eta\rangle[\,],\ [\,]) \vdash ([\,],\ \ \varepsilon)$
2. $(\langle/\eta\rangle\ ,\ [\,]) \vdash ([\,],\ \ \varepsilon)$
3. $(\langle\eta\rangle[\,],\ [s]) \vdash ([s],\ \langle\eta\rangle[\,])$
4. $(\langle/\eta\rangle\ ,\ [s]) \vdash ([s],\ \langle/\eta\rangle)$
5. $(\langle\eta\rangle[s],\ \_) \vdash ([s],\ \langle\eta\rangle[s])$

(a) Diagonal filter.

1. $(\langle\eta\rangle[\,],\ \ [\,]) \vdash (\ \ [\,],\ \ \varepsilon)$
2. $(\langle/\eta\rangle\ ,\ \ [\,]) \vdash (\ \ [\,],\ \ \varepsilon)$
3. $(\langle\eta\rangle[c],\ \ \_) \vdash (\ \ [c]\,|\,\_,\ \langle\eta\rangle[c])$
4. $(\langle\eta\rangle[\,],\ \_\,|\,\gamma) \vdash ([\,]\,|\,\_\,|\,\gamma,\ \langle\eta\rangle[\,])$
5. $(\langle/\eta\rangle\ ,\ \_\,|\,\gamma) \vdash (\ \ \gamma,\ \ \langle/\eta\rangle)$

(b) Vertical filter.

Fig. 10. Configuration-based transitions for diagonal and vertical filter transducers.

make the filters more expressive than SPEX transducers. Initially, there is an empty annotation on their stacks.

The transition rules of the diagonal filter read as follows. If only empty annotations have been received (stated by the empty annotation as the only stack entry), then no message is let through. As soon as the stack consists of a non-empty annotation, all subsequent messages are let through. Finally, in case the received node has a non-empty annotation ($[s] \neq [\,]$), then it is sent through and the annotation becomes the stack content.

The vertical filter uses its stack to remember the smallest depth of a received node with a non-empty annotation. Therefore, only if the stack consists of an empty annotation, then the opening and closing tags of nodes with empty annotations are not let through.

VII. Complexity

This section gives polynomial upper bounds for the complexity of Forward Core XPath query evaluation against XML streams. The polynomial lower bounds for *in-memory* evaluation of Core XPath are given by [26]. References [27], [28] give memory lower bounds for the evaluation of queries from a large XPath fragment against non-recursive streams. For queries with closure axes and predicates, [28] shows that any streaming evaluation algorithm must use at least $\Omega(\mathsf{CONCUR}(D,Q))$ memory space, where $\mathsf{CONCUR}(D,Q)$ is the maximum number of candidates for the evaluation of Q against the stream D at any instant. In worst case, this number is the number of stream nodes and hence the entire stream has to be buffered.

In the remainder we consider that the query has size q (i.e., number of steps inside and outside predicates) and p outermost predicates (i.e., predicates not included in other predicates); the tree conveyed in the stream has depth d, breadth b, size s, and number of nodes n. We also use $c = \mathsf{CONCUR}(D,Q)$.

We next present the space and time combined complexities for the evaluation of queries from five XPath fragments. The rationale behind choosing these fragments is given by the various sizes of annotations created during query evaluation and by the lack or need to buffer stream fragments. The syntactical characterization of these fragments is given in Fig. 11. All fragments contain non-closure axes, nodetests, and all boolean and set operators.

Following [18], [28] we only consider the problem of deciding whether each node is in the answer set or not. Note that SPEX fully supports the XPath semantics in that it outputs the query answer and in stream order. This additional computational step can take quadratic time in the stream size, because the answer size can be quadratic in the input stream size.

Theorem 7.1: SPEX has time complexity $O(q \times s \times a_i)$ and space complexity $O((q \times d + p \times c) \times a_i)$ for XPath_i and a_i as defined in Fig. 11 ($1 \leq i \leq 5$).

Proof: [Sketch] For all our XPath fragments the following three properties hold. First, the size of a transducer network for a query is linear in the query size (see the compilation in Section IV). Second, the size of annotations present in the stream influences both the time and the space complexities of query evaluation. Third, a transducer stack can store at most d annotations, as shown next.

Fragment	predicates	closure axes	annotation size a_i
XPath_1	none	+	$O(1)$
XPath_2	vertical	–	$O(1)$
XPath_3	vertical	+	$O(d)$
XPath_4	horizontal	+	$O(d+b)$
XPath_5	diagonal	+	$O(n)$

Fig. 11. XPath fragments and corresponding SPEX annotation sizes.

An annotation can only follow an opening tag in the stream. For each received annotation, a transducer pushes an annotation onto the stack and for each closing tag an annotation is popped from the stack. A stack can have at most d entries (=annotations), for there can be at most d opening tags encountered in the stream before one of their closing tags is received.

To process an XML stream, a transducer network needs then time linear in q and space linear in q and d. The time and space complexities depend also on the size of annotations created during processing, as discussed next.

XPath_1 queries have no predicates, thus (a) there are no candidates to buffer, and (b) the only used annotations are the full and the empty annotations, both of constant size.

XPath_{2-5} queries can have predicates. The evaluation of queries with predicates can require a buffer of maximum size $p \times c \times a_i$: there are c candidates at any instant and for each candidate we keep at most a_i annotations for each of the scopes that nest the head transducer (in total p such scopes).

XPath_2 queries have no closure axes, thus there is no need to union annotations, and each transducer matches at a fixed depth in the stream. This makes that only three annotations are used during processing, namely [ ], [0], and [1], all of constant size.

XPath_{3-5} queries can have closure axes and predicates. Depending on the predicate types, the unions of annotations is bounded by d (for vertical predicates), $d+b$ (for horizontal predicates), or n (for diagonal predicates). ■

Remark 7.2: XPath_1 queries have no predicates and therefore $p = 0$. Thus the space complexity for XPath_1 becomes $O(q \times d)$. If XPath_{3-5} queries are restricted to only have closure axes, one can show that the annotations can be represented as continuous intervals of integers, where the biggest integer is bounded by d (for vertical predicates), $d+b$ (for horizontal predicates), or n (for diagonal predicates). Then one only needs $\log(d)$, $\log(d+b)$, or $\log(n)$ bits respectively, to represent an annotation. □

VIII. Experiments

The polynomial combined complexity of SPEX is verified by an extensive experimental evaluation conducted on a prototype implementation of SPEX in Java (Sun JRE 1.5) on a Pentium 1.5 GHz with 500 MB under Linux 2.4.

XML Streams. We consider the effect of varying the stream size s on the evaluation time for two XML stream sets. The first set [29] provides real-life XML streams of up to 21 million nodes and depth up to 36. We used in the experiments the small XML documents region, nation, courses, sigmod, part, and orders (with sizes from 1 KB to 5MB), and as medium to large XML documents nasa, lineitem, treebank, dblp, and protein (with sizes from 23 to 680 MB). The second set provides synthetic XML streams with a slightly more complex structure that allows more precise variations in the workload parameters [17]. The synthetic data is generated from information about processes running on computer networks and corresponds to the output of the Linux (SUSE) command "ps -elfH" in XML.

Queries. For each considered XML document we generated queries using its DTD. This led to queries that express tree navigation compatible to the document structure defined by the DTD. The query generation was tuned with the query size q (which means the number of location steps) and several probabilities: p_{nextSibl}, p_+, p_λ, and p_* for next-sibling, closure axes, predicates, and wildcard nodetest respectively. For example, a path query has $p_\lambda = 0$. For each parameter setting, 10–50 queries were tested.

The query generation algorithm works as follows. We first construct a graph representing the input DTD, where each node represents a (possibly optional) token. Child-edges and sibling-edges are created from a node x to a (not necessarily distinct) node y if y appears in the content model of x in the DTD, or if y can appear as a next-sibling in a document instance of the DTD respectively. To generate a query, we start with the graph node corrsponding to the given top token and decide to navigate along a child-edge or a sibling-edge depending on the probability p_{nextSibl}; depending on p_+, we decide to jump several edges of the same type and depending on p_λ we decide to create a predicate with the steps generated using the same procedure (predicates with simple paths or boolean connectors have equal probabilities). Depending on p_*, we take as nodetest the current token, or the wildcard '*'. This procedure ends when the total number of generated steps reaches the bound q. If the procedure ends before reaching the desired number of steps, then additional dummy steps self::* are appended.

Note that this algorithm can generate queries that yield empty answers. This is because a DTD defines a set of different XML documents and a generated query may have non-empty answers on a subset of these documents, which may not contain the document from our dataset. In our tests, about 5% of the queries have empty answers.

Scalability. Scalability results are presented for stream and query sizes. In both 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 that of the stream or query sizes. Fig. 12 emphasizes the theoretical results: The 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 for the synthetic data due to its more complex structure.

Varying the query characteristics. Fig. 13(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 nextSibl and child are comparable.

The **memory usage** is almost constant over the full range of the previous tests. Cf. Fig. 13(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 the Runtime Java package).

Reducing the stream traffic. All previous tests show results for "naive" SPEX, i.e., SPEX without the filters described in Section VI. Fig. 14 shows how these filters affect the evaluation time. The phase_2 filters (vertical and horizontal) improve the evaluation time up to 3 times for our tests using queries, whose sizes range from 5 to 1000, cf. Fig. 14(a). The same figure shows also that, for small XML streams, our evaluation strategy is in average five times slower than the mere parsing of the XML stream[1], if phase_2 is used, 10 times slower if phase_1 (diagonal filters only) is used, and 15 times slower for naive SPEX. Using phase_2, an increase in the query size q tends to have little to constant influence on the evaluation time. This result is explained by the fact that an increase in the query size leads often to an increase in its selectivity, thus supporting the rationale for employing filters. The same explanation applies to Fig. 14(b), where the increase of the closure probability (p_+) makes the queries less selective and leads to a less effective gain achieved by the filters.

Non-polynomial behavior of other engines. Work independent of ours confirms experimentally the polynomial complexity of SPEX versus the exponential complexity of the XPath engine Xalan-Java 2.6 [30] for queries that involve closure axes [31]. Xalan, as also other engines [9]–[11], have exponential query complexity because they lack efficient set-at-a-time processing: given a path and a set of context nodes, Xalan computes the set of nodes reachable via the given path from each context node independently of the other context nodes, although the computed sets can overlap. To ensure correct answer without duplicates and in stream order, Xalan unions the computed sets, sorts the result, and removes the duplicates. This operation can take exponential space in the query size, because for each step the set of nodes reachable from any node can be linear in the stream size. The engines [9]–[11] are based on automata that can have an exponential number of states to encode that at any instant the current node can be matched by several query steps.

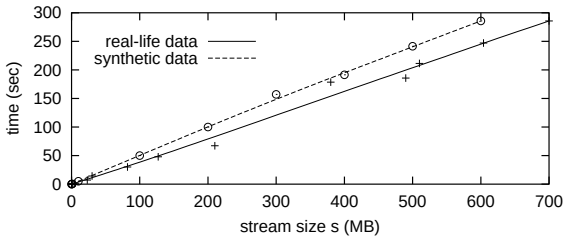
IX. Related Work

There is a large amount of work in the field of XPath evaluation against XML streams. We look next at the characteristics of the most known processors through SPEX glasses.

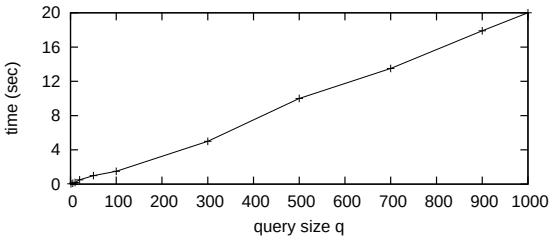
In the context of publish-subscribe or event notification systems, the XML stream needs to be filtered by a large number of simple forward queries. Engines like [9], [10], [12] assume the stream partitioned into small XML documents (up to thousands of elements per XML document). Except of [12], they perform query matching with exponential query complexity. Recently, [27] gives lower bounds for the query matching problem in the case of two fragments of XPath (Univariate XPath and Structural Subsumption-free XPath) and non-recursive streams. It also gives a matching algorithm, whose space is close to these bounds.

The query answering engines XSQ [11] and HAOS [13] and TwigM [32] are the closest in spirit to SPEX and deserve closer

[1]We used the Crimson SAX parser available at `http://xml.apache.org/crimson/`.

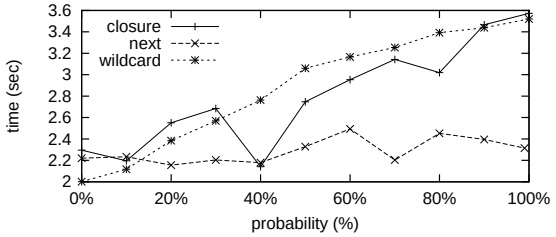


(a) Varying stream size s ($q = 10$, $3 \leq d \leq 32$)

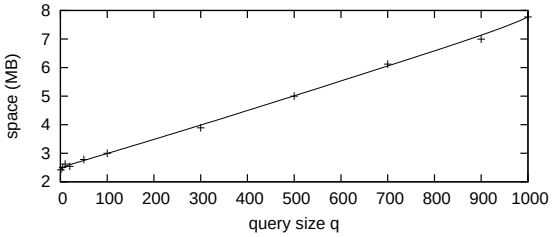


(b) Varying query size q ($s = 244$ kB, $d = 32$)

Fig. 12. Scalability ($p_* = p_+ = p_{\text{nextSibl}} = p_\lambda = 0.5$)

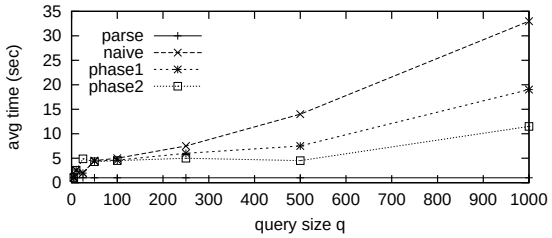


(a) Effect of p_*, p_+, and p_{nextSibl}

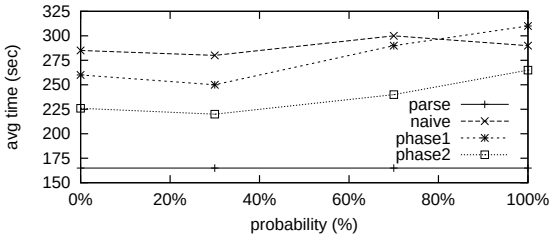


(b) Effect of varying query size q

Fig. 13. If not varied, $s = 244$ kB, $d = 32$, $q = 10$, $p_* = p_+ = p_{\text{nextSibl}} = p_\lambda = 0.5$



(a) Varying query size q ($s = 450$ kB)



(b) Effect of p_+ ($s = 700$ MB, $q = 10$)

Fig. 14. Effect of filters. If not varied, $p_* = p_+ = p_{\text{nextSibl}} = 0.5$

inspection. XSQ supports queries limited to child and descendant axes and unnested predicates with at most one step. It can compile one query into an exponential number of pushdown transducers augmented with queues that are gathered into a hierarchical deterministic pushdown transducer. XSQ can perform an exponential number of operations per stream message, even for non-recursive streams.

HAOS supports child and descendant axes and their symmetrical reverse axes parent and ancestor. A query is compiled into a DAG structure where nodes are XPath nodetests and edges are XPath axes. The reverse axes are rewritten similar to [16] using rewrite rules of [15]. The evaluation is based on the incremental construction of a matching structure consisting of mappings of nodes from the DAG query to nodes from the tree conveyed in the input stream. This evaluation approach is similar to the standard tree pattern evaluation algorithm, presented, e.g., in [33], though the latter constructs the matching structure bottom-up in the data tree, whereas the former constructs the structure top-down, as imposed by the stream's sequence. All answers of the query are accumulated and are delivered after processing the entire stream. Thus, no progressive processing is performed. An answer is determined uniquely by exactly one matching of each query node, and all these matchings are accumulated until the end of the processing. SPEX constructs also a matching structure updated constantly on the arrival of new stream messages and distributed on the stacks of its transducers. However, at any time this structure contains only sufficient information to determine the next answers, and previous matchings that are not needed anymore for possible new answers are dropped. This way, the memory footprint of SPEX remains lower than that of HAOS.

Recently, [32] presented an efficient query answering engine called TwigM for an XPath fragment with child and descendant axes and predicates (thus strictly weaker than our XPath fragment). The experimental evaluation reported in [32] shows that TwigM scales very well when compared to XSQ [11] and XMLTK [10].

XSM [34] is a streaming engine also based on networks of transducers. Unlike a SPEX pushdown transducer, an XSM transducer has buffers with random access and several read and write pointers. Our SPEX transducers clearly show that an efficient implementation of any XPath forward axis does not need the expressiveness of such complex XSM transducers.

XSM can evaluate queries consisting of steps with descendant axis and nodetests different from wildcard, value-based joins, and XQuery static element constructors against XML streams with non-recursive structure definition. Recall that the key feature of SPEX is the efficient processing of *structural* joins (the XPath forward axes) on *arbitrary* XML streams. We do not see any straightforward extension of XSM to cope with XPath axes and arbitrary XML streams. For the (rather trivial) XPath fragment and XML streams supported by both XSM and SPEX, we note that both engines become very similar. Due to the severe

restriction on the input XML streams, SPEX transducers for a step child$^+$::η, with a nodetest η different from wildcard, can only match at most one node along any path from the root to a leaf. Thus the SPEX transducers do not need stacks. Then, like for XSM, we can compose all transducers of a network into a single finite transducer [35].

X. Conclusion

This article describes SPEX, a streamed and progressive evaluation of XPath queries against XML streams. The streamed aspect of SPEX resides in the sequential (as opposed to random) access to the XML stream. SPEX is progressive because it delivers the query answers as soon as possible. Queries are compiled into networks of deterministic transducers that process XML streams with polynomial combined complexity. Experiments confirm the scalability of SPEX.

Acknowledgment

I thank Fatih Coskun for the implementation of the current version of SPEX.

References

[1] N. Koudas and D. Srivastava, "Data stream query processing: A tutorial," in *Proc. Int. Conf. Very Large Databases (VLDB)*, 2003, p. 1149.
[2] NASA, *JPL Sensor Webs Project*, http://sensorwebs.jpl.nasa.gov, 2004.
[3] ——, *XML Group Resources Page*, http://xml.gsfc.nasa.gov, 2004.
[4] *Traffic and Traveller Information (TTI) – TTI messages via traffic message coding – Part 1: Coding protocol for Radio Data System – Traffic Message Channel (RDS-TMC)*, International Standard Organization (ISO), 2003, http://www.iso.org.
[5] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Trans. on Networking (TON)*, vol. 9, no. 3, pp. 280–292, 2001.
[6] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith, "Hancock: A language for extracting signatures from data streams," in *Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD)*, 2000, pp. 9–17.
[7] D. Rogers, J. Hunter, and D. Kosovic, "The TV-trawler project," *Journal of Imaging Systems and Technology*, pp. 289–296, 2003.
[8] Z. G. Ives, A. Y. Halevy, and D. S. Weld, "An XML query engine for network-bound data," *VLDB Journal*, vol. 11, no. 4, pp. 380–402, 2002.
[9] M. Altinel and M. J. Franklin, "Efficient filtering of XML documents for selective dissemination of information," in *Proc. Int. Conf. Very Large Databases (VLDB)*, 2000, pp. 53–64.
[10] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML streams with deterministic automata," in *Proc. Int. Conf. Database Theory (ICDT)*, 2003, pp. 173–189.
[11] F. Peng and S. S. Chawathe, "XPath queries on streaming data," in *Proc. Int. Conf. Management of Data (SIGMOD)*, 2003, pp. 431–442.
[12] C.-Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi, "Efficient filtering of XML documents with XPath expressions," in *Proc. Int. Conf. Data Eng. (ICDE)*, 2002, pp. 235–244.
[13] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski, "Streaming XPath processing with forward and backward axes," in *Proc. Int. Conf. Data Eng. (ICDE)*, 2003, pp. 455–466.
[14] C. Koch and S. Scherzinger, "Attribute grammars for scalable query processing on XML streams," *VLDB Journal*, 2007, to appear.
[15] D. Olteanu, H. Meuss, T. Furche, and F. Bry, "XPath: Looking forward," in *Proc. EDBT Workshop XMLDM*, 2002, pp. 109–127.
[16] D. Olteanu, T. Furche, and F. Bry, "Evaluating complex queries against XML streams with polynomial combined complexity," in *Proc. British National Conf. on Databases (BNCOD)*, 2004, pp. 31–44, prel. version in technical report PMS-FB-2003-15, University of Munich, Feb. 2003.
[17] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel, "The XML stream query processor SPEX," in *Proc. Int. Conf. Data Eng. (ICDE)*, 2005, demonstration.
[18] G. Gottlob, C. Koch, and R. Pichler, "Efficient algorithms for processing XPath queries," in *Proc. Int. Conf. Very Large Databases (VLDB)*, 2002, pp. 95–106.
[19] J. Clark and S. DeRose, "XML path language (XPath) version 1.0," World Wide Web Consortium," W3C Recommendation, 1999.
[20] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon, "XQuery 1.0: An XML query language," World Wide Web Consortium," Working Draft, 2002.
[21] J. Clark, "XSL transformations (XSLT) version 1.0," World Wide Web Consortium," W3C Recommendation, 1999.
[22] D. C. Fallside and P. Walmsley, "XML-Schema," World Wide Web Consortium," W3C Recommendation, 2001, http://www.w3.org/XML/Schema.
[23] S. DeRose, R. D. Jr., P. Grosso, E. Maler, J. Marsh, and N. Walsh, "XML pointer language (XPointer)," World Wide Web Consortium," W3C Recommendation, 2002, http://www.w3.org/TR/xptr/.
[24] D. Olteanu, "Forward node-selecting queries over trees," *ACM Trans. Database Systems (TODS)*, March 2007, to appear.
[25] E. Gurari, *An Introduction to the Theory of Computation*. Computer Science Press, 1989.
[26] G. Gottlob, C. Koch, and R. Pichler, "The complexity of XPath query evaluation," in *Proc. ACM Symposium Principles of Database Systems (PODS)*, 2003, pp. 179–190.
[27] Z. Bar-Youssef, M. Fontoura, and V. Josifovski, "On the memory requirements of XPath evaluation over XML streams," in *Proc. ACM Symposium Principles of Database Systems (PODS)*, 2004, pp. 177–188.
[28] ——, "Buffering in query evaluation over XML streams," in *Proc. ACM Symposium Principles of Database Systems (PODS)*, 2005, pp. 216–227.
[29] G. Miklau, *XML Data Repository*, Univ. of Washington, 2003, http://www.cs.washington.edu/research/xmldatasets.
[30] *Xalan-Java Version 2.6*, Apache Project, 2005, http://xml.apache.org/xalan-j/index.html.
[31] A. Berlea, "Event-driven evaluation of grammar queries," Technische Universität München, Tech. Rep., 2005, available online http://wwwbib.informatik.tu-muenchen.de/infberichte/2005/TUM-I0513.ps.gz.
[32] Y. Chen, S. Davidson, and Y. Zheng, "An efficient XPath query processor for XML streams," in *Proc. Int. Conf. Data Eng. (ICDE)*, 2006.
[33] G. Miklau and D. Suciu, "Containment and equivalence of a fragment of XPath," *Journal of the ACM*, vol. 51, no. 1, pp. 2–45, 2004.
[34] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou, "A transducer-based XML query processor," in *Proc. Int. Conf. Very Large Databases (VLDB)*, 2002, pp. 227–238.
[35] C. Choffrut and K. Culik II, "Properties of finite and pushdown transducers," *SIAM Journal of Computing*, vol. 12, no. 2, pp. 300–315, 1983.

Dan Olteanu received the diploma degree in computer science from the Polytechnic University of Bucharest in 2000 and completed his diploma thesis in the Caravel project at INRIA-Rocquencourt. After receiving the PhD degree in computer science from the University of Munich in 2005, he joined the newly created database and information systems group at Saarland University. His research interests are in incomplete information, data integration, and XML query processing.