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

Ludwig
Maximilians–
Universität
München
LMU

An Evaluation of Regular Path Expressions with Qualifiers against XML Streams

Dan Olteanu, Tobias Kiesling, François Bry

Technical Report, Computer Science Institute, Munich, Germany
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-2002-12, May 2002 (Revised Dec. 2002)

Abstract

This paper presents SPEX, a streamed and progressive evaluation of regular path expressions with XPath-like qualifiers against XML streams. SPEX proceeds as follows. An expression is translated in linear time into a network of transducers, most of them having 1-DPDT equivalents. Every stream message is then processed once by the entire network and result fragments are output on the fly. In most practical cases SPEX needs a time linear in the stream size and for transducer stacks a memory quadratic in the stream depth. Experiments with a prototype implementation point to a very good efficiency of the SPEX approach.

I. Introduction

Querying data streams is a field of growing importance motivated by applications such as real time measurements (e.g. monitoring the number of passing objects on a channel for traffic routing) and continuous services which select informations from a continuous stream of data (e.g. stock exchange or meteorology data). For a selective dissemination of information (SDI), streams have to be filtered according to complex requirements, specified as queries, before being distributed to the subscribers [1], [17]. To integrate data over the Internet, particularly from sources with low troughputs, it is desirable to progressively process the data before the full stream is retrieved [2], [3]. The data streams considered in such applications can be infinite (or, more precisely, unbound) and consist of structured messages. Such messages are conveniently modeled with XML. Message selection, i.e. queries, are naturally expressed using regular path expressions with qualifiers [4].

This paper describes a model called SPEX for a streamed and progressive evaluation of regular path expressions against XML streams. Streamed evaluation means that a data stream is not completely buffered, progressive processing means that results are streamed and delivered on the fly. The SPEX model handles not only simple regular path expressions, but also regular path expressions with qualifiers like those of XPath [5].

Some approaches to streamed query evaluation consider fixed size *windows* on the data stream for restricting the query evaluation to parts of the input data [6]. This way, input streams of unbound sizes can be processed without storing more data than specified by the window. However, this is at the cost of returning incorrect and/or incomplete answers. In SPEX, windows are *not* considered (although windows could easily be combined with SPEX), i.e. SPEX performs an exact evaluation. SPEX does store parts of the input data stream in memory only if their

appartenence to the query result is not yet determined.

The approach proposed here enjoys the following attractive features:

• The translation of regular path expressions with qualifiers into a SPEX transducer network takes a time linear in the size of the input expression.

• The evaluation of a SPEX network is performed in one pass over the stream and requires for every transducer stack a number of entries bounded in the depth d of the stream. In most practical cases it takes a time linear in the stream size and uses formulas with a size bounded in d. Without qualifiers or closure steps the size of a formula is constant. The evaluation of expressions with qualifiers on n wildcard closure steps can require formulas with size exponential in the number of such steps, i.e. d^n. The last transducer takes care of outputting ordered result and in the worst case it needs a memory linear in the stream size. However, it buffers messages only if their membership in the result can not be decided based on the stream fragment already processed.

• The computational power needed by a transducer from a SPEX network, except its last transducer, is within the 1-DPDT class. Note that this is the lowest bound for the computational power needed for querying XML streams with regular path expressions [7]. The output transducer needs the computational power of a Turing machine.

• Experiments with a prototype implementation point to a very good efficiency of SPEX. The prototype supports also other XPath navigational capabilities, i.e. following and preceding, and node-identity joins. Compared with existing XPath processors, SPEX overcomes them in most of the medium-sized scenarios and scales acceptable in cases when the other processors can not handle huge data, e.g. ≥ 1 GB [7]. The prototype was tested also against application-generated infinite streams and proved stable in cases where the depth of the tree conveyed in the stream is bounded.

The paper is organized as follows. Section II shortly recalls the XML stream data model and regular path expressions. The SPEX evaluation model is introduced in Section III. In this section, SPEX transducers for each expression construct are first specified. Then the translation of regular path expressions with qualifiers into a SPEX transducer network is explained. The computational power of a SPEX transducer is investigated in Section IV. Complexity results are given in Section V. Section VI compares the efficiency of a SPEX prototype with other XML query implementations that construct in-memory representations of the streams. The migration

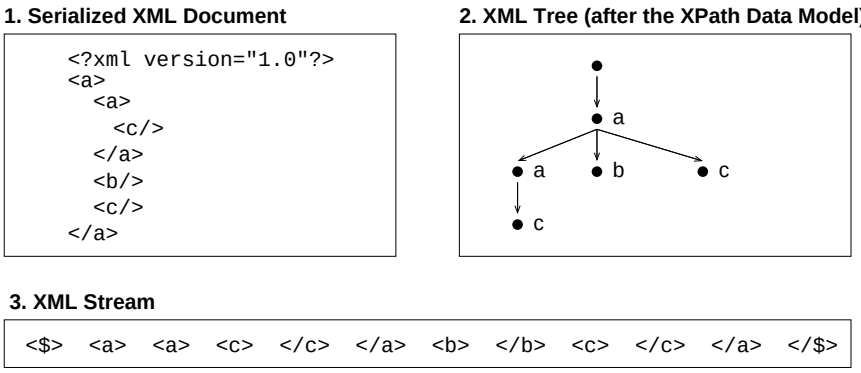
to conjunctive queries with regular path expressions is outlined in Section VII. Related work is summarized in Section VIII. Section IX is a conclusion.

II. Preliminaries

1. XML Streams

In this paper, an XML stream denotes a sequence of document messages (or events). A stream carries parent-child relationships between document messages, i.e. it specifies structural information. Streaming an XML document corresponds to a traversal of the XML document in so-called document tree order, i.e. a depth-first left-to-right traversal of the document tree. At the start (end, resp.) of the document a start-document message `<$>` (end-document message `</$>`, resp.) is placed on the stream. Such streams are similar to the event streams generated by the Simple API for XML (SAX) [8]. Figure 1 illustrates on a simple example the streams considered in this paper.

Fig. 1 Three Representations of an XML Document



For reasons of conciseness, XML specificities such as attributes, namespaces, processing instructions and comments, are not considered. The necessary extensions are technical, but not difficult.

2. Regular Path Expressions with Qualifiers (RPEQ)

The query languages for semistructured data and XML have the ability to express queries in form of path expressions [4]. A path expression is a query composed of steps that express a navigation through the tree associated with the XML data. Using path expressions one might express (direct or indirect) parent-child relationships on the nodes of a document tree.

A regular path expression is a regular expression [9] over an alphabet V with the operators

closure (*), concatenation (.), and union (|). Here, V is the set of labels of the nodes of an XML document tree (cf. Figure 1). The evaluation of a regular path expression E against a stream representing an XML document D is the sequence of nodes in the data tree of D that are reachable from the root of this tree by paths conforming to E.

A natural enhancement for regular path expressions, as encountered in the XML query language XPath [5], is the addition of qualifiers. Here, a qualifier $[E]$ is defined in terms of a regular path expression E. The evaluation of the qualifier $[E]$ returns the truth value *true,* if the evaluation of the regular path expression E returns a non-empty set of nodes, and *false,* otherwise.

The regular path expression with qualifiers, short rpeq, considered in this paper, are described by the following grammar:

rpeq ::= ϵ | *label* | *label* + | *label* * | (*rpeq* | *rpeq*) | (*rpeq* . *rpeq*) | *rpeq* ? | *rpeq* [*rpeq*]

The symbol *label* stands for a node label or for the wildcard (_) that matches every node labels. Note that the operators * and ? can be defined as follows using other operators: *label** is equivalent to (*label*+ | ϵ) and *rpeq*? is equivalent to (*rpeq* | ϵ). E.g. _*.a[b]._*.c is a rpeq which selects the c labeled nodes that are descendant of an a labeled node having at least one b labeled child. In this rpeq, [*b*] is a qualifier.

The language of rpeq covers the XPath fragment with no other steps than the forward steps `child` and `descendant` and no other qualifiers than structural qualifiers. Backward steps like `ancestor` and `parent` are expressible with rpeq, since, as shown in [10], they are expressible in the above-mentioned XPath fragment.

III. The SPEX Evaluation Model

This section describes an evaluation of regular path expressions with qualifiers (rpeq) based on transducers. For each rpeq construct, a so-called SPEX transducer is specified. An rpeq is translated into a SPEX network consisting of interconnected SPEX transducers.

1. SPEX Transducer Network

A finite-state transducer (FST) is an abstract computing machine with a finite state control, a read-only input tape, and a write-only output tape. A *pushdown transducer* (1-PDT) is a finite-state transducer augmented with a pushdown store [9]. A SPEX pushdown transducer, as introduced here, is similar to a conventional deterministic pushdown transducer (1-DPDT),

except that it does not have accepting states and that it has two stacks, a "condition stack" and a "depth stack", instead of only one, i.e. it is a 2-DPDT. The condition stack is used for keeping track of condition formulas on which query results depend, and the depth stack used for counting the document tree level of elements.

Definition 1 (SPEX Transducer). A SPEX transducer

$$T = (\mathcal{Q}, \Sigma, \Omega, \Gamma_{depth}, \Gamma_{cond}, q_0, \delta)$$

is a 2-DPDT with two stacks, called *condition* and *depth* stacks, such that:

- $\mathcal{Q}$ is a finite set of states,
- Σ is the input alphabet,
- Ω is the output alphabet,
- Γ_{depth} is the alphabet of the depth stack,
- Γ_{cond} is the alphabet of the condition stack,
- $q_0 \in \mathcal{Q}$ is the initial state, and
- δ is the transition function

$$\delta : \Sigma \times \mathcal{Q} \times \Gamma_{depth} \times \Gamma_{cond} \rightarrow \mathcal{Q} \times (\Gamma_{depth} \cup \{\epsilon\}) \times (\Gamma_{cond} \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}).$$

Each transition depends on the next input symbol, the current state, the top of the depth and condition stacks. It possibly changes the state, the top of both stacks, and the output tape.

The transitions of different SPEX transducers are specified in the following sections by means of configurations. A configuration of a SPEX transducer is a triple $\phi \in \mathcal{Q} \times \Gamma^*_{depth} \times \Gamma^*_{cond} = \Phi$. The transitions are specified as relations from $\Sigma \times \Phi$ to $\Phi \times \Omega^*$, i.e. as relations from configurations under input symbols to new configurations with output symbols.

SPEX transducers can be modified by adding extra input and/or output tapes, as shown in Section III.6. For conciseness, the notation T^{in}_{out} is used to refer to a SPEX transducer with input tapes $in = in_1, \ldots, in_n$ and output tapes $out = out_1, \ldots, out_m$. In Sections III.3 through III.7, several SPEX transducers are introduced. Note that their computational power is that of 1-DPDT as argued in Section IV. Section III.8 introduces a more complex transducer that has the computational power of a general 2-DPDT, hence of a Turing machine [11].

A SPEX network consists in a collection of SPEX transducers with their interconnections. A connection is used to transmit different kinds of messages between transducers: Document

messages, activation messages, and condition determination messages. The last two kinds of messages serve only the communication between transducers.

Definition 2 (SPEX messages). There are three kinds of messages in a SPEX network:
Document messages of the forms `<a>` or `</a>` indicate the beginning or the end of elements in an XML stream.
Activation messages of the form $[f]$ activate transducers with a *condition formula* f, i.e. make transducers return results when f becomes `true`. A condition formula consists of conjunctions and/or disjunctions of *condition variables*.
Condition determination messages of the form $\{c,v\}$ signal the value v (true or false) of a condition variable c.

A condition variable represents an instance of a qualifier. Most SPEX transducers, e.g. the child, closure, split, and join transducers, do not process condition formulas, but just receive, store and send them as they are. In contrast, the variable-filter SPEX transducer decomposes formulas into a stream of condition variables and filters out or check some of the variables.

Definition 3 (SPEX Network). A SPEX Network SN of degree n is a tuple

$$SN = (DM, AM, CM, G)$$

where

- DM, the set of document messages, is the input and output alphabet of the network,
- AM is the set of activation messages that may be passed in the network,
- CM is the set of condition determination messages that may be passed in the network,
- $G = (N, E)$ is a directed acyclic graph (DAG) with a set of nodes $N = \{T_1, \ldots, T_n\}$ of cardinality n and with set of edges E.

A node T_i $(1 \leq i \leq n)$ of G is a SPEX transducer

$$T_i = (\mathcal{Q}^i, \Sigma, \Omega, \Gamma^i_{depth}, \Gamma_{cond}, q^i_0, \delta^i)$$

where $\Sigma = \Omega = DM \cup AM \cup CM$, and where $\Gamma_{cond} = \{f \mid [f] \in AM\}$, Γ^i_{depth}, q^i_0, δ^i and Q^i depend on the type of the SPEX transducer. An edge of G represent a connection between two SPEX transducers: $(T_i, T_j) \in E$, if an output tape of T_i is connected with, i.e. is identical to, the input tape of T_j.

Using the short-hand notation for transducers, a graph G defining a SPEX network can be represented as the set $\{\mathsf{T}_1{}^{in_1}_{out_1}, \ldots, \mathsf{T}_n{}^{in_n}_{out_n}\}$ of its transducers, i.e. of its nodes: The connections between the transducers, i.e. the edges of G, can be determined as follows:

$$(\mathsf{T}_i{}^{in_i}_{out_i}, \mathsf{T}_j{}^{in_j}_{out_j}) \in E \Leftrightarrow out_i \cap in_j \neq \emptyset$$

The SPEX networks considered below are directed acyclic graphs (DAG) with one root (or source) and one sink.

The translation of a regular path expression with qualifiers into a SPEX network is given in Section III.9 after the SPEX transducers corresponding to the *rpeq* constructs are introduced. SPEX transducers have transitions (left implicit in the following specifications) that forward document messages along the SPEX network without processing them, in case no other (explicitely specified) transition applies.

2. Input Transducer: IN^{in}_{out}

A SPEX network for a given query is formalized as a DAG with one source and one sink. One source means querying a single stream. One sink means collecting results for a single query. Extensions of SPEX networks are possible e.g. allowing multiple sources, i.e. querying several streams, or allowing multiple sinks, i.e. evaluating several queries.

The source of a SPEX network is an *input transducer* which has the task of sending an activation message on the start document message and of forwarding one document message at a time. After a message reaches the network's sink, the next message is forwarded by the input transducer into the network, thus ensuring that at any time there is only one message in the network.

3. Child Transducer: $\mathsf{CH}^{in}_{out}(l_m)$

A child transducer, cf. Figure 2, represents a label selection for, say, label l_m, i.e. it selects <l_m> document messages at a specific depth (of the unmaterialized document tree) . Such a transducer tries to match only document messages that are direct children of the activating document message, i.e. that have a tree level of $n+1$, where n is the tree level of the activating document message. The document messages, that can be matched by a transducer, define its *match scope*.

For matching only direct children of the activating document messages, the transducer must

Fig. 2 Transitions of child transducer $\mathsf{CH}^{in}_{out}(l_m)$ for matching l_m document messages

$\Gamma^{\mathtt{ch}}_{\mathtt{depth}}$ = {m,l} $\mathcal{Q}^{\mathtt{ch}}$ = {waiting,matching,activated1,activated2} $q_0^{\mathtt{ch}}$ = waiting

1	([f] , (waiting , α, β))	$\vdash$	((activated1, α, $f\|\beta$)), ϵ	)
2	(<l> , (waiting , α, β))	$\vdash$	((waiting , l$\|\alpha$, β)), <l>	)
3	(</l> , (waiting , l$\|\alpha$, β))	$\vdash$	((waiting , α, β)), </l>	)
4	(</l> , (waiting , m$\|\alpha$, β))	$\vdash$	((matching , α, β)), </l>	)
5	(<l> , (activated1, α, β))	$\vdash$	((matching , l$\|\alpha$, β)), <l>	)
6	([f] , (matching , α, β))	$\vdash$	((activated2, α, $f\|\beta$)), ϵ	)
7	(<l_m> , (matching , α, $f\|\beta$))	$\vdash$	((waiting , m$\|\alpha$, $f\|\beta$)), [f];<l_m>	)
8	(<l_n> , (matching , α, β))	$\vdash$	((waiting , m$\|\alpha$, β)), <l_n>	)
9	(</l> , (matching , l$\|\alpha$, $f\|\beta$))	$\vdash$	((waiting , α, β)), </l>	)
10	(</l> , (matching , m$\|\alpha$, $f\|\beta$))	$\vdash$	((matching , α, β)), </l>	)
11	(<l_m> , (activated2, α, $f_1\|f_2\|\beta$))	$\vdash$	((matching , m$\|\alpha$, $f_1\|f_2\|\beta$)), [f_2];<l_m>)	
12	(<l_n> , (activated2, α, β))	$\vdash$	((matching , m$\|\alpha$, β)), <l_n>	)
13	({c,v}, (*any_state* , α, β))	$\vdash$	((*any_state* , α, update(c,v,β)), {c,v}	)

keep track of the tree level, i.e. the depth reached in the (unmaterialized) document tree, and distinguish between the levels of direct children and the other tree levels; these levels are marked with different symbols (m for match and l for level) on the depth stack.

The child transducer can be activated in two contexts: While waiting for being activated, as shown in transition (1) of Figure 2, or while trying to match, as result of a previous activation (6). Hence, it is able to match in several matching scopes originating from different activating messages.

Note that the transitions of SPEX transducers consider only one input symbol. In Figure 2 (and later in Figure 3), two extra states activated1 (1,5) and activated2 (6,11,12) are introduced to accommodate the case where two consecutive input messages have to be treated, i.e. when a transducer gets activated by receiving an activation message and a document message which led to the activation.

Example III.1. Consider the evaluation of the *rpeq* $a.c$ against the stream from Figure 1. For a and c *rpeq* constructs two child transducers are created. The second transducer, T_2, has as input tape the output tape of the first transducer, T_1. The first component in the network is an input transducer IN and its output tape is the input tape of T_1. The transitions of transducers

Fig. 3 Transitions of closure transducer $\mathsf{CL}^{in}_{out}(l_m)$ for matching l_m document messages

Γ^{cl}_{depth} = {l,s,ns,e} $\mathcal{Q}^{cl}$ = {waiting,matching,activated1,activated2} q_0^{cl} = waiting

1	([*f*]	, (waiting	,	α,	β)) $\vdash$ ((activated1,		α,	$f \mid \beta$),	ϵ	)
2	(<*l*>	, (waiting	,	α,	β)) $\vdash$ ((waiting	, l$\mid\alpha$,		β),	<*l*>	)
3	(</*l*>	, (waiting	, l$\mid\alpha$,		β)) $\vdash$ ((waiting	,	α,	β),	</*l*>	)
4	(</*l*>	, (waiting	, e$\mid\alpha$,		β)) $\vdash$ ((matching	,	α,	β),	</*l*>	)
5	(<*l*>	, (activated1,		α,	β)) $\vdash$ ((matching	, s$\mid\alpha$,		β),	<*l*>	)
6	([*f*]	, (matching	,	α,	β)) $\vdash$ ((activated2,		α,	$f \mid \beta$),	ϵ	)
7	(<l_m>	, (matching	,	α,	$f \mid \beta$)) $\vdash$ ((matching	, l$\mid\alpha$,		$f \mid \beta$),	[*f*];<l_m>	)
8	(<l_n>	, (matching	,	α,	β)) $\vdash$ ((waiting	, e$\mid\alpha$,		β),	<l_n>	)
9	(</*l*>	, (matching	, l$\mid\alpha$,		β)) $\vdash$ ((matching	,	α,	β),	</*l*>	)
10	(</*l*>	, (matching	, ns$\mid\alpha$,		$f \mid \beta$)) $\vdash$ ((matching	,	α,	β),	</*l*>	)
11	(</*l*>	, (matching	, s$\mid\alpha$,		$f \mid \beta$)) $\vdash$ ((waiting	,	α,	β),	</*l*>	)
12	(<l_m>	, (activated2,		α,	$f_1 \mid f_2 \mid \beta$)) $\vdash$ ((matching	, ns$\mid\alpha$,		$f_1 \vee f_2 \mid f_2 \mid \beta$),	[f_2];<l_m>	)
13	(<l_n>	, (activated2,		α,	β)) $\vdash$ ((matching	, ns$\mid\alpha$,		β),	<l_n>	)
14	({c,v},	(*any_state*	,	α,	β)) $\vdash$ ((*any_state*	,	α, update(c,v,β)),		{c,v}	)

T_1 and T_2 for processing the entire stream are shown in Figure 4.

Fig. 4 The sequence of transitions for child transducers T_1 and T_2 from Example III.1

$\mathsf{CH}^{in}_{out}/DM$	<$>	<a>	<a>	<c>	</c>				<c>	</c>		</$>
T_1	1,5	7	2	2	3	3	2	3	2	3	4	9
T_2	2	1,5	8	2	3	4	8	4	7	4	9	3

When a start-document message <$> is encountered, IN sends to T_1 an activation message with the formula true, cf. transitions (1) and (5) of T_1 (T_1: 1,5). The first <a> document message is matched by T_1 that sends an activation message to T_2 (T_1: 7). After matching, T_1 enters in the waiting state, and remembers the tree level where it is supposed to match by pushing an m symbol on its depth stack. The transducer T_2 is activated and prepared for matching (T_2: 1,5). The second <a> document message is not matched by T_2, that enters in the waiting state, and remembers in the same manner the tree level where it is supposed to match (T_2: 8). The next two document messages, <c> and </c>, are not matched, as both transducers are waiting to reach their marked tree level. At the level of <b> document message, T_2 tries to match again (T_2: 8). When T_2 encounters the <c> document message, at the same level with <b>, and corresponding

to the tree level marked on its depth stack with the `m` symbol, it matches and creates a candidate for the result (T_2: 7). Exiting the match scopes is done for T_2 after encountering the last but one document message, and for T_1 after encountering the end-document message (T_1, T_2: 9).

4. Closure Transducer: $CL^{in}_{out}(l_m)$

A *closure transducer*, cf. Figure 3, implements the logic of the positive closure `+` on a label l_m, i.e. it selects nested <l_m> document messages. Kleene closure `*` can be derived from positive closure, as described in Section II.2.

The match scope of this transducer contains all nested document messages that are descendants of the activating document message in the tree, and have the same label l_m. Like the child transducer, the closure transducer can be activated while waiting for being activated (1), or while trying to match, as result of a previous activation (6). This transducer is able to match for several nested activations at the same time, hence it can have several match scopes. Keeping track of the tree levels corresponding to the window of each match scope is done by using three symbols on the depth stack: `s`, for marking the beginning of a match scope that is not nested inside other scopes, `ns` for marking the beginning of a match scope that is nested inside other scopes, and `e` for marking the end of a match scope. The further symbol `l` is used for marking the tree levels that do not fall in one of the special cases mentioned above. A nested match scope can appear if the transducer was already activated and is in the process of matching. The reason for differentiating between the match scopes is that leaving the first kind of scope leads to changing the `matching` state to the `waiting` state (11), while a nested scope leaves the transducer in the same `matching` state (10), since there remains at least one other match scope, which contained the current one.

An interesting property of the closure transducer is that, for a nested scope, a disjunction of the formula received and of the topmost formula of its condition stack is pushed on the condition stack (12). This reflects the fact that a closure transducer can match due to both nesting and nested scopes, while being inside the nested scope. Note, that such a disjunction can be normalized by removing multiple occurrences of the same conjuncts. In this way, a formula contains at most one reference to a condition variable.

Example III.2. Consider the evaluation of the *rpeq* `a+.c+` against the stream from Figure 1. For `a+` and `c+` *rpeq* constructs two closure transducers are created. The second transducer, T_2, has

as input tape the output tape of the first transducer, T_1. The first component in the network is an input transducer IN and its output tape is the input tape of T_1. The transitions of both transducers for processing the entire stream are shown in Figure 5.

Fig. 5 The sequence of transitions for closure transducers T_1 and T_2 from Example III.2

$\mathsf{CL}^{in}_{out}/DM$	<$>	<a>	<a>	<c>	</c>				<c>	</c>		</$>
T_1	1,5	7	7	8	4	9	8	4	8	4	9	11
T_2	2	1,5	6,13	7	9	10	8	4	7	9	11	3

When a start-document message <$> is encountered, IN sends to T_1 an activation message with a `true` formula. When the first <a> is encountered, T_1 matches and sends to T_2 an activation message with the formula from the top of its condition stack (i.e. `true`), cf. transition (7) of T_1 (T_1: 7). Hence, T_2 is awaken (T_2: 1,5). The next document message is also an <a>, that is matched by T_1 (T_1: 7), causing a nested match scope for T_2 (T_2: 6). The transducer T_2 also tries to match the <a> in the previous scope. It does not match, but it enters again in a matching state, due to the second scope (T_2: 13). When <c> is encountered in the stream, T_2 matches and creates a candidate for the result, which depends on the formula from the top of its condition stack (T_2: 7). Note that this matching is due to the second scope. After the next </c> and document messages are processed, the second scope is closed and both transducers are still in their first scope (T_1: 9, T_2: 10). The document message closes the scopes for both transducers (T_1, T_2: 8), and opens them again (T_1, T_2: 4), in a new attempt to match <a> and <c> document messages, respectively. A <c> document message is encountered, which closes the scope window of T_1 (T_1: 8). This document message is matched by T_2 and becomes part of a result candidate (T_2: 7). Processing the </c> and the <$> document messages, respectively, leads to the ending of the match scope of T_2 and T_1, respectively (T_1, T_2: 11).

5. Qualifier Transducers

A qualifier instance can be seen as a condition variable, on which candidates for the result depend. Consider for example the expression $\_^*.a[b]$, where $[b]$ is a qualifier. For each a in the document that is matched, an instance of the qualifier $[b]$ is considered. This instance has a truth value, which represents the fulfillment of the qualifier for a node a.

In a SPEX network, variables are created for each qualifier and – depending on the fulfillment of their related qualifier instances – evaluated to a truth value. A qualifier adds to a network

transducers for creating variables and for handling determination of the values of variables.

III.5.1 Variable-Creator Transducer: $\mathsf{VC}^{in}_{out}(q)$

A *variable-creator transducer* is responsible for creating a condition variable c for each instance of the qualifier q. Its transitions are shown in Figure 6. The creation of variables is triggered by a received activation message $[f]$, when it outputs a new activation message, consisting of a conjunction between the received boolean formula and the newly created variable (1). This transducer also invalidates the created variable c when its scope has been left, by sending a condition determination message $\{c, \texttt{false}\}$ (4).

Fig. 6 Transitions of variable-creator transducer $\mathsf{VC}^{in}_{out}(q)$

$\Gamma^{\text{vc}}_{\text{depth}}$ = {1,s} $\mathcal{Q}^{\text{vc}}$ = {working,activate} q_0^{vc} = working

1 ([f] , (working , α, β)) ⊢ ((activate, α, $c|\beta$), [$f \wedge c$])
2 (<l> , (working , α, β)) ⊢ ((working , l|α, β), <l>)
3 (</l> , (working , l|α, β)) ⊢ ((working , α, β), </l>)
4 (</l> , (working , s|α, $c|\beta$)) ⊢ ((working , α, β), {c,false};</l>)
5 (<l> , (activate, α, β)) ⊢ ((working , s|α, β), <l>)
6 ({c,v}, (*any_state*, α, β)) ⊢ ((*any_state*, α, β), {c,v})

III.5.2 Variable-Filter Transducer: $\mathsf{VF}^{in}_{out}(q\pm)$

A *variable-filter transducer* is defined for a qualifier q and is sensitive to condition variables created for that qualifier. There are two distinct variable-filter transducers, a *negative variable-filter transducer* $\mathsf{VF}^{in}_{out}(q-)$ for dropping the variables created for a qualifier q, and a *positive variable-filter transducer* $\mathsf{VF}^{in}_{out}(q+)$ for dropping everything else but those variables.

III.5.3 Variable-Determinant Transducer: VD^{in}_{out}

A *variable-determinant transducer* for a given qualifier q provides determination messages for the instances of that qualifier. If an instance c of q is satisfied, then a determination message that assigns a `true` value to c is sent (1), as described in Figure 7.

Each instance c of q is received within an activation message from a positive variable-filter $\mathsf{VF}^{in}_{out}(q+)$, which filters out from condition formulas all other variables that do not belong to q. Every instance c of q that reaches this transducer via an activation message is satisfied.

Fig. 7 Transitions of variable-determinant transducer VD^{in}_{out}

$$\Gamma^{\mathtt{vd}}_{\mathtt{depth}} = \emptyset \quad \mathcal{Q}^{\mathtt{vd}} = \{\mathtt{working}\} \quad q_0^{\mathtt{vd}} = \mathtt{working}$$

$$1\ ([c]\ ,\ (\mathtt{working}\ ,\ \alpha,\ \beta)) \vdash ((\mathtt{working}\ ,\ \alpha,\ \beta),\ \{c,\mathtt{true}\})$$
$$2\ (\{c,v\},\ (any\_state,\ \alpha,\ \beta)) \vdash ((any\_state,\ \alpha,\ \beta),\ \epsilon\ )$$

6. *Split and Join Transducers:* $SP^{in}_{out_1,out_2}$, JO^{in_1,in_2}_{out}

Split and join transducers allow more complex SPEX networks than simple transducer chains, by providing support for parallel stream processing and synchronization primitives, encoded at the level of transducer transitions.

The *split transducer* has two output tapes. Its task is to forward every received message to both of the output tapes. The *join transducer* has two input tapes. Its task is to collect the messages coming on its input tapes and to send (some of) them to the output. Its behaviour is similar to an `AND`-gate, where a signal level (here a document message) is produced at the output when on both inputs that signal level is encountered. Because of this particularity, this transducer plays an important role in the synchronization of the network, ensuring that both network branches before the join transducer are expected to finish before other components, situated after the join transducer, can continue their work. A join transducer is only sensitive to document messages, hence activation and condition determination messages can pass unconditioned through it without an explicit treatment.

In the specification given for a split transducer in Figure 8 the right-hand side of the transition is a 5-tuple, as it shows both of the output tapes. The right-hand side of the transition relation for a join transducer, as presented in Figure 9 is shortened to a 2-tuple consisting of the new state and the output tape. The left-hand side of transitions is also changed to show the current state and both input tapes.

Fig. 8 Transitions of split transducer $\mathsf{SP}^{in}_{out_1,out_2}$

$$\Gamma^{\mathtt{sp}}_{\mathtt{depth}} = \emptyset \quad \mathcal{Q}^{\mathtt{sp}} = \{\mathtt{working}\} \quad q_0^{\mathtt{sp}} = \mathtt{working}$$

$$1\ (any\_symbol,\ (\mathtt{working},\ \alpha,\ \beta)) \vdash ((\mathtt{working},\ \alpha,\ \beta),\ any\_symbol,\ any\_symbol)$$

Note that a split (a join, resp.) transducer with n output (input, resp.) tapes can be simulated

Fig. 9 Transitions of join transducer $\mathsf{JO}^{in_1,in_2}_{out}$

$\Gamma^{\mathtt{jo}}_{\mathtt{depth}} = \emptyset \quad \mathcal{Q}^{\mathtt{jo}} = \{\mathtt{none}, \mathtt{left}, \mathtt{right}\} \quad q^{\mathtt{jo}}_0 = \mathtt{none}$

1	(<l>	, <d>	, none) ⊢	(none	, <l>	)
2	(<l>	, [f]	, none) ⊢	(left	, [f]	)
3	(<l>	, {c,v}	, none) ⊢	(left	, {c,v}	)
4	([f]	, <d>	, none) ⊢	(right	, [f]	)
5	({c,v}	, <d>	, none) ⊢	(right	, {c,v}	)
6	([f]	, {c,v}	, none) ⊢	(none	, [f];{c,v}	)
7	({c,v}	, [f]	, none) ⊢	(none	, [f];{c,v}	)
8	([f_1]	, [f_2]	, none) ⊢	(none	, [f_1];[f_2]	)
9	({c_1,v_1},	{c_2,v_2},	none) ⊢	(none	, {c_1,v_1};{c_2,v_2}	)
10	(ϵ	, [f]	, left) ⊢	(left	, [f]	)
11	(ϵ	, {c,v}	, left) ⊢	(left	, {c,v}	)
12	(ϵ	, <d>	, left) ⊢	(none	, <l>	)
13	([f]	, ϵ	, right) ⊢	(right	, [f]	)
14	({c,v}	, ϵ	, right) ⊢	(right	, {c,v}	)
15	(<l>	, ϵ	, right) ⊢	(none	, <l>	)

by $n-1$ transducers with 2 output (input, resp.) tapes.

7. *Union Transducer:* UN^{in}_{out}

A *connector transducer* has the task of creating a condition formula from two formulas it receives. Such a new formula is a disjunction or a conjunction of the formulas received.

A *union transducer* UN^{in}_{out} is a connector that creates a disjunction between formulas from two activation messages it receives. As described in Figure 10, it stores a condition formula from a received activation message $\mathtt{[f_1]}$ in (1), and then a second condition formula $\mathtt{f_2}$ in (2), when it also sends an activation message with a disjunction of them ($\mathtt{[}f_1 \vee f_2\mathtt{]}$). If it receives only one activation message, then it forwards it (3).

At the query language level, a union transducer conveys the semantics of a union operation. The union of two sets of document messages is computed here by looking at one message from one or both sets at a time. If a document message `<l>` is part of a set, then the transducer that matched the document message sends an activation message followed by the matched document message (1,3). The same document message may become part of the second set only at the same time, since only one document message is passed at a time through the network. In this case,

Fig. 10 Transitions of union transducer UN^{in}_{out}

$$\Gamma^{\mathtt{un}}_{\mathtt{depth}} = \emptyset \quad \mathcal{Q}^{\mathtt{un}} = \{\mathtt{waiting},\ \mathtt{activate}\} \quad q_0^{\mathtt{un}} = \mathtt{waiting}$$

$$\begin{aligned}
&1\ ([f_1]\ ,\ (\mathtt{waiting}\ ,\ \alpha,\ \ \beta)) \vdash ((\mathtt{activate},\ \alpha,\ f_1 \mid \beta),\ \epsilon\ \ )\\
&2\ ([f_2]\ ,\ (\mathtt{activate},\ \alpha,\ f_1 \mid \beta)) \vdash ((\mathtt{waiting}\ ,\ \alpha,\ \ \beta),\ [f_1 \vee f_2])\\
&3\ (\langle l\rangle\ \ ,\ (\mathtt{activate},\ \alpha,\ f_1 \mid \beta)) \vdash ((\mathtt{waiting}\ ,\ \alpha,\ \ \beta),\ [f_1];\langle l\rangle)\\
&4\ (\{c,v\},\ (\mathit{any_state},\ \alpha,\ \ \beta)) \vdash ((\mathit{any_state},\ \alpha,\ \ \beta),\ \{c,v\}\ \ )
\end{aligned}$$

an activation message for that document message depending on the disjunction of the condition formulas of both received activation messages is output (1, 2).

Note that a connector transducer has only one input tape. For simulating a set operation, it is assumed to appear in a network after a join transducer, which can interleave messages coming from separate branches. Also the problem of removing duplicates for the union operation is solved by the join transducer, which allows only unique document messages to pass.

8. Output Transducer: OU^{in}_{out}

The sink of a SPEX network is an *output transducer*, which processes results, i.e. query answer candidates. Its task is to identify and store candidates, to evaluate conditions formulas so as to decide whether a result candidate is a result, and to output results in document order.

The output transducer acts as follows: If an activation message is on its input tape, a new candidate is created, which consists of the range of document messages starting with the first received one, e.g. `<1>`, and ending with its corresponding end document message, e.g. `</1>`. If a condition determination message $\{c, v\}$ is on the input tape, then the transducer updates the candidate formulas with the assignment v for c. In such a case, it is possible that the truth value of a formula attached to a candidate is determined and thus the candidate becomes part of the result, or is dropped. However, if a candidate is part of the result, the preservation of the document order can impose that it must be stored until all earlier candidates are determined.

The output transducer is more complex than the other introduced transducers, as it needs random access to result candidates and condition formula stacks.

9. Translation from Regular Path Expressions to SPEX Networks

The translation of a regular path expression with qualifiers into a SPEX network is done statically, before starting the process of query evaluation, and is given by means of a denotational semantics, specified as a function $\mathcal{C}$ shown in Figure 11. The function $\mathcal{C}$ is defined by induction on the structure of regular path expressions, as introduced in Section II. This is possible due to the compositional nature of regular path expressions and SPEX networks.

$\mathcal{C}$ maps a regular path expression, a given SPEX network configuration σ, and its output tape t to an updated SPEX network and an output tape, the new network being formed from the initial network and the network representation of the given expression. A SPEX network is represented as a set of interconnected SPEX transducers, cf. Definition 3.

Fig. 11 A denotational semantics for regular path expressions with qualifiers

$\mathcal{C}$: Expression $\rightarrow$ (Network, Tape) $\rightarrow$ (Network, Tape)

$$\mathcal{C}[\![(rpeq_1 \mid rpeq_2)]\!](\sigma,t) = \{(\sigma_4,t_6) \mid \sigma_1 = \{\sigma, \mathsf{SP}^t_{t_1,t_2}\}, (\sigma_2,t_3) = \mathcal{C}[\![rpeq_1]\!](\sigma_1,t_1), (\sigma_3,t_4) = \mathcal{C}[\![rpeq_2]\!](\sigma_2,t_2), \sigma_4 = \{\sigma_3, \mathsf{JO}^{t_3,t_4}_{t_5}, \mathsf{UN}^{t_5}_{t_6}\}\}$$

$$\mathcal{C}[\![(rpeq_1\,.\,rpeq_2)]\!](\sigma,t) = \mathcal{C}[\![rpeq_2]\!](\mathcal{C}[\![rpeq_1]\!](\sigma,t))$$

$$\mathcal{C}[\![rpeq?]\!](\sigma,t) = \{(\sigma_3,t_4) \mid \sigma_1 = \{\sigma, \mathsf{SP}^t_{t_1,t_2}\}, (\sigma_3,t_3) = \mathcal{C}[\![rpeq]\!](\sigma_1,t_1), \sigma_3 = \{\sigma_2, \mathsf{JO}^{t_2,t_3}_{t_4}\}\}$$

$$\mathcal{C}[\![label\ ^*]\!](\sigma,t) = \{(\sigma_3,t_4) \mid \sigma_1 = \{\sigma, \mathsf{SP}^t_{t_1,t_2}\}, (\sigma_2,t_3) = \mathcal{C}[\![label\ ^+]\!](\sigma_1,t_1), \sigma_3 = \{\sigma_2, \mathsf{JO}^{t_2,t_3}_{t_4}\}\}$$

$$\mathcal{C}[\![\ label\ ]\!](\sigma,t) = (\{\sigma, \mathsf{CH}^t_{t_1}(\ l\ )\}, t_1)$$

$$\mathcal{C}[\![\ label\ ^+]\!](\sigma,t) = (\{\sigma, \mathsf{CL}^t_{t_1}(\ label\ )\}, t_1)$$

$$\mathcal{C}[\![rpeq_1\ \texttt{[}\ rpeq_2\ \texttt{]}]\!](\sigma,t) = \mathcal{C}[\![\ \texttt{[}\ rpeq_2\ \texttt{]}\ ]\!](\mathcal{C}[\![rpeq_1]\!](\sigma,t))$$

$$\mathcal{C}[\![\ \texttt{[}\ rpeq\ \texttt{]}\ ]\!](\sigma,t) = \{(\sigma_3,t_7) \mid \sigma_1 = \{\sigma, \mathsf{VC}^t_{t_1}(q), \mathsf{SP}^{t_1}_{t_2,t_3}\}, (\sigma_2,t_4) = \mathcal{C}[\![rpeq]\!](\sigma_1,t_2), \sigma_3 = \{\sigma_2, \mathsf{VF}^{t_4}_{t_5}(q+), \mathsf{VD}^{t_5}_{t_6}, \mathsf{JO}^{t_3,t_6}_{t_7}\}\}$$

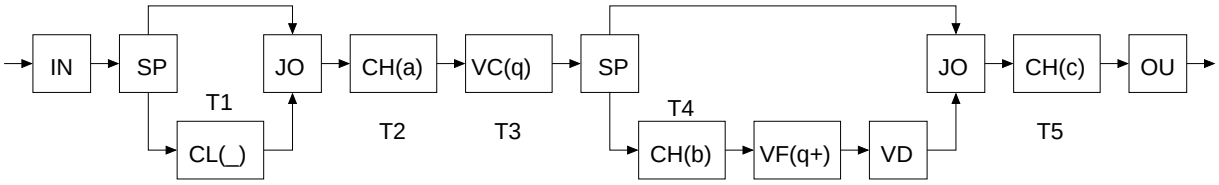
The input and output transducers are added to a SPEX network after the internal components are set up by means of $\mathcal{C}$. Hence, a SPEX network corresponding to a regular path expression with qualifiers *rpeq* is $(\{\sigma, \mathsf{OU}^{t_1}_{t_2}\} \mid (\sigma,t_1) = \mathcal{C}[\![rpeq]\!](\mathsf{IN}^{in}_t, t))$.

10. Complete Example

Assume the regular path expression with qualifiers $\_*.a[b].c$ is to be evaluated against the stream of Figure 1. The corresponding SPEX network is shown in Figure 12 where each box represents a transducer. The transition sequence of each transducer is given in Figure 13.

The following explanations are focused on handling qualifier instances, i.e. condition variables,

Fig. 12 SPEX Network for $\_*.a[b].c$



and result candidates depending on them. The start-document message `<$>` is processed by the input transducer by writing to its output tape an activation message with a `true` formula (`[true]`), as there are no qualifiers before the input transducer. T_1 and T_2 receive the activation message, push the formula on their condition stack and forward the start-document message. The other transducers are not activated, they just forward the start document message and count the tree level. For the start-element message `<a>`, T_1 and T_2, respectively, succeed to match, and they activate T_2 and T_3, respectively. T_3 on its turn activates T_4 and T_5 with a conjunction between the formula within the received activation message and a newly created condition variable co_1. The second document message `<a>` is matched again by T_1, T_2 and T_3 and a new condition variable co_2 is created by T_3. T_5 matches the first `<c>` document message, and proposes a new result candidate $candidate_1$ containing this message. This candidate depends on variable co_2, which is not yet determined. When the first `</a>` is encountered, T_3 sets co_2 to `false` by sending a condition determination message $\{co_2, \texttt{false}\}$. This is done since the match scope of the inner `<a>` has been left, and no `<b>` has been encountered for satisfying the qualifier. As co_2 is determined to have a `false` value, the output transducer can discard $candidate_1$. When the first `<b>` document message is encountered, the first instance of the qualifier is satisfied, i.e. the variable-determinant transducer `VD` sends a condition determination message $\{co_1, \texttt{true}\}$. The first candidate that depends on co_1, i.e. $candidate_2$, is created after processing the second `<c>` document message. This candidate is directly sent to output, since the formula it depends on is determined and has a `true` value.

IV. Computational Power of SPEX Transducers

The following theorem gives a lower bound for the computational power needed by a *rpeq* evaluator for querying well-formed XML documents.

Theorem IV.1 (Querying well-formed XML Documents with *rpeq*). *The computational*

Fig. 13 The sequence of transitions for SPEX transducers from Figure 12

T/DM	<$>	<a>	<a>	<c>	</c>				<c>	</c>		</$>
T_1	1,5	7	7	7	9	9	7	9	7	9	9	11
T_2	1,5	6,11	6,11	6,12	10	10	6,12	10	6,12	10	10	9
T_3	2	1,5	1,5	2	3	4	2	3	2	3	4	3
T_4	2	1,5	6,12	8	4	13,10	7	4	8	4	9	3
T_5	2	1,5	6,12	7	4	13,10	13,8	4	7	4	9	3

power needed for querying well-formed XML documents with rpeq is at least that of pushdown automata with one stack (1-PDA).

Proof. (sketched) The problem of querying an XML document with *rpeq* consists in (1) identifying parts of document that are considered for the result, and (2) constructing the result from those identified parts. In the following, the first subproblem is analyzed. An *rpeq* evaluator is a recognizer of well-formed XML documents, where the language describing a document and recognized by the evaluator is enriched with constraints derived from the semantics of *rpeq*. It is shown that there exists a regular path expression that needs the computational power of 1-PDA.

Consider the regular path expression a that selects all nodes with label a (<a>...</a>) that are children of the document root (<$>...</$>). A language that conveys the same semantics as the given regular path expression and has any well-formed XML document as instance is

$$L(a) = \texttt{<\$>}\ (\ \texttt{<}x\texttt{>}^m\texttt{</}x\texttt{>}^m\ (\ \texttt{<}a\texttt{>}\ \texttt{<}y\texttt{>}^n\ \texttt{</}y\texttt{>}^n\ \texttt{</}a\texttt{>}\ )*\ )*\ \texttt{</\$>},$$

where $m, n \geq 0$, y can be any tag, and x can be any tag but a. By the pumping lemma for regular languages [9], the language L is not regular, hence it can not be accepted by a finite state automaton, but rather by a pushdown automaton. Hence, the *rpeq* query subproblem needs at least the computational power of 1-PDA. By extension, the *rpeq* querying problem needs at least the same computational power.

Informally, the reason behind this needed computational power is that inside an a matching node there can be an arbitrary number n of nested subnodes, which can also be a nodes. Keeping track of their proper nesting in order to match nodes with label a only appearing at the level immediate below the root implies the use of a pushdown store. □

In the following, it is shown that the computational power of the SPEX evaluation model is the very lower bound of the previous theorem. I.e. in terms of computational power, the SPEX

model is optimal. Recall that a pushdown transducer is in fact a pushdown automaton with one write-only output tape, i.e. pushdown transducers and pushdown automata with one write-only output tape have the same computational power.

Theorem IV.2 (Computational Power of SPEX Transducers). *The computational power of the child (closure, union, qualifier, resp.) SPEX transducer is that of deterministic pushdown transducers with one stack (1-DPDT).*

Proof. SPEX transducers are introduced in Section III.3 through III.7 as deterministic pushdown transducers with 2 stacks (2-DPDT). The child, closure and variable-creator transducers use both stacks, i.e. the depth and the condition stack. Note, however, that the use of the stacks is done in a highly synchronized manner, and there are no transitions that push onto one stack and pop from the other one, so as to simulate a Turing machine [11]. The only operations done at a time on the stacks are: pushing from both stacks or popping from both stacks, pushing or popping from one stack, while the other one remains unchanged. Hence, both stacks can be simulated by one stack, where an entry is represented by a data structure composed of two entries, from the depth stack and the condition stack, respectively.

The union transducer uses only one entry on the condition stack for storing temporary a condition formula, hence it is in 1-DPDT class. The variable-filter, variable-determinant, split, and join transducers do not use stacks at all, hence they are in FST (finite state transducer) class, which make them also part of 1-DPDT class. □

The only transducer that uses two stacks in an unsynchronized manner is the output transducer. Its computational power is that of 2-DPDT class, hence that of Turing machines [11].

V. Complexity Results

In the following, an XML stream is described by its size s and the depth d of its (unmaterialized) associated document tree, $rpeq(n)$ denotes a regular path expression with qualifiers of length n, S_t (T_t, resp.) denotes the space (time, resp.) a SPEX transducer t needs for processing an expression $rpeq(n)$ against an XML stream.

Lemma V.1 (SPEX Network Construction). *The translation time and the degree (cf. Definition 3) of a SPEX network for $rpeq(n)$ is linear in n.*

Proof. The expression $rpeq(n)$ is translated into a network of SPEX transducers, as presented in Section III.9. The number of SPEX transducers created for each expression construct is constant,

hence the degree of the network corresponding to $rpeq(n)$ is linear in n. The time for adding each transducer to the network is constant, hence the translation time is linear in n. □

The space complexity for a SPEX transducer depends on the maximum sizes of its depth and condition stacks. The entries of a depth stack are of a constant size and a depth stack can have at most d entries, since it counts the tree levels conveyed in the stream. Therefore, the space needed by a depth stack is linear in d. The time for pushing and popping a depth stack entry is constant. Hence, the time needed for managing a depth stack while processing the entire stream is linear in s. Below, only results concerning the condition stacks are given.

As described in Section III, an activation message carries a condition formula. An *rpeq* expression can match nested document messages at most d times. For each qualifier, condition instances are created which are represented by condition variables. The number of undetermined condition variables at a time in the system is at most d and a condition formula can refer at a time to at most d condition variables. A closure transducer can create formulas that represent disjunctions of formulas; a variable-creator transducer can create formulas that represent conjunctions of other formulas.

The size σ of a boolean formula. First, the size σ of a boolean formula is analyzed. Here it is considered only the size of a boolean formula in the case of a rpeq without unions and multiple qualifiers on a single step.

The results can be extended to rpeq with unions in the following way: if an rpeq has m union terms, then the size σ of a boolean formula ϕ' is $\sigma(\phi') = m * \sigma(\phi)$, where $\sigma(\phi)$ is the maximum size of a formula created during the evaluation of a single union term. In the same way, the results can be extended to multiple qualifiers on a single step.

Regarding the supported rpeq language fragment, the following cases can be distinguished:

- the language fragment without qualifers, i.e. rpeq*.

In this case, there can be only a single boolean formula in the condition stacks, i.e. `true`. For this language fragment, the condition stacks can be even dropped.

$\sigma(\phi) = 1$.

- the language fragment without closure, but with qualifiers, i.e. rpeq$^{[]}$.

In this case, a boolean formula can consist only in conjunctions between boolean variables. More specifically, a boolean formula can consist of at most $min(n, d)$ boolean variables, where `n` is the number of qualifiers in the expression and `d` is the depth of the tree conveyed in the stream.

$\sigma(\phi) = min(n, d)$.

• the language fragment with closure and qualifiers, i.e. rpeq$^{*,[]}$.

Consider the following general form for a rpeq$^{*,[]}$: $rpe_1[q_1].rpe_2[q_2]. \ldots .rpe_n[q_n]$, where every rpe_i contains a closure step. The boolean variables created for $[q_i]$ are generally denoted v_i, e.g. v_i^1 is the first variable created for the qualifier $[q_i]$. There are n_i variables created for $[q_i]$ that correspond to the number of active matchings at a time of rpe_i. In the general case, n_i is bounded in the depth d of the stream.

On the stacks of the first rpe_2 closure step there can be formulas representing disjunctions of variables v_1, at most n_1. The biggest formula ϕ_2 looks like: $\phi_2 = v_1^1 \vee v_1^2 \ldots \vee v_1^{n_1}$, $\sigma(\phi_2) = n_1$. On the stacks of next steps from rpe_2 the previous formula can appear in every entry, thus a stack entry has an n_1 size.

Going further, on the stacks of the first rpe_3 closure step there can be formulas representing a conjunction between a disjunction of all variables of the first qualifier and a disjunction of all variables of the second qualifier: $\phi_3 = (v_1^1 \vee v_1^2 \ldots \vee v_1^{n_1}) \wedge v_2^1 \vee \ldots \vee (v_1^1 \vee v_1^2 \ldots \vee v_1^{n_1}) \wedge v_2^{n_2}$, $\sigma(\phi_3) = (n_1 + 1) \times n_2$. This formula can appear in every entry, thus a stack entry in this case has an $(n_1 + 1) \times n_2$ size.

We can conclude that a result candidate depends on a formula ϕ that can have the size:
$\sigma(\phi) = \Sigma_{j=1}^{n}(\Pi_{i=j}^{n}(n_i))$. Therefore, $\sigma(\phi) = O(d^n)$.

Remark V.1. In the cases in which two distinct *rpe* steps do not match the same stream messages the following assumption holds: $\Sigma_{i=1}^{n}(n_i) \leq d$, i.e. the sum of the number of active matchings at a time for all steps is bounded in the stream depth d. This happens when rpe_1 to rpe_n match altogether sequentially in depth.

In this simplified case, a better result can be provided. Consider again a formula from the stack of an rpe_3 step, i.e. $\phi_3 = (v_1^1 \vee \ldots \vee v_1^{n_1}) \wedge v_2^1 \vee \ldots \vee (v_1^1 \vee \ldots \vee v_1^{n_1}) \wedge v_2^{n_2}$. It is easy to see that $\phi_3 = (v_1^1 \ldots \vee v_1^{n_1}) \wedge (v_2^1 \vee \ldots \vee v_2^{n_2})$. In this case, $\sigma(\phi_3) = n_1 + n_2$. Furthermore, a formula ϕ_n from the stacks of rpe_n steps is: $\phi_n = (v_1^1 \vee \ldots \vee v_1^{n_1}) \wedge \ldots \wedge (v_{n-1}^1 \vee \ldots \vee v_{n-1}^{n_{n-1}})$. Thus, $\sigma(\phi) = \Sigma_{i=1}^{n}(n_i) \leq d$.

Lemma V.2 (Space and Time Complexity of SPEX Transducers). *The space and time needed for querying a stream with size s and depth d by*

1. *a child or closure transducer is $S_{CH} = S_{CL} = O(d \times \sigma)$, $T_{CH} = T_{CL} = O(\sigma \cdot s)$;*
2. *a variable-creator is $S_{VC} = O(d)$, $T_{VC} = O(\sigma \cdot s)$;*

3. a variable-creator or union transducer is $S_{UN} = O(\sigma)$, $T_{UN} = O(\sigma \cdot s)$;

4. an input, variable-filter, condition-determinant, split or join transducer is $S_t = O(1)$, $T_t = O(\sigma \cdot s)$, *where* $t \in \{IN, VF, CD, SP, JO\}$;

5. and an output transducer is $S_{OU} = O(\sigma \cdot s)$, $T_{OU} = O(\sigma \cdot s)$.

Proof. 1. A condition stack allows entries, represented by condition formulas, which can have size σ. There can be d entries on a condition stack, due to the maximum of d nested activations. Hence, the space needed by a child and closure transducer is: $S_{CH} = S_{CL} = O(d \times \sigma)$.
The processing time needed for one incoming message is determined by the received formulas within activation messages. A transducer needs time σ for copying a formula to and from its stack. For the entire stream, the processing time is linear in s: $T_{CH} = T_{CL} = O(\sigma \cdot s)$.
2. A variable-creator for a qualifier q instantiates for each received activation a new condition variable with q stamp. Each condition variable is stored on an entry on the condition stack, and the condition stack can have at most d entries, which is the maximum number of nested activations. Hence, the space needed by a variable-creator transducer is: $S_{VC} = O(d)$.
The processing time needed for one incoming message is determined by the received formulas within activation messages. A transducer needs time σ for copying a formula to and from its stack. For the entire stream, the processing time is linear in s: $T_{VC} = O(\sigma \cdot s)$.
A union transducer needs one entry on the condition stack for storing a formula: $S_{DC} = O(\sigma)$.
The processing time of one message from the input stream implies the time for handling also the possible related activation message, i.e. the time for copying it on the stack, which is σ, and the time for creating a disjunction of two formulas with conjuncts duplicate elimination, which is also linear in σ. For the entire stream, $T_{DC} = O(\sigma \cdot s)$.
3. Input, variable-filter, condition-determinant, split and join transducers do not use their push-down store, hence their space requirement is constant: $S_t = O(1)$. Copying an activation message of size σ from input to output requires σ time. Since there can be as many activation messages as document messages, the time needed for copying all the s activation messages is: $T_t = O(\sigma \cdot s)$.
The input transducer does not encounter activation messages: $T_{IN} = O(s)$.
4. The output transducer needs space for candidates and for their formulas. A formula has size σ, a candidate can be of size linear in s, e.g. the entire stream is a candidate that depends on a qualifier instance that can be evaluated only at the end of the stream: $S_{OU} = O(\sigma \cdot s)$.
Managing the candidates and their related formulas implies time for copying a formula for each

candidate. There can be s candidates, alltogether of size s, and for each of them a formula of size σ is copied on a store: $T_{OU} = O(\sigma \cdot s)$.

□

Theorem V.1 (Querying XML Streams with SPEX Networks). *The space S_{net} and the time T_{net} needed by a SPEX network* **net** *for querying a stream with size s and document depth d are $S_{net} = O(max(d \times \sigma, s))$ and $T_{net} = O(\sigma \cdot s)$.*

Proof. The space and time used by a SPEX network `net` for querying a stream with size s and depth d is given by the sum of the space and the time, respectively, needed by its components. Hence, the space and time complexity is: $S_{\texttt{net}} = O(s)$ and $T_{\texttt{net}} = O(\sigma \cdot s)$. □

Under the assumption that $d \ll s$, i.e. the document depth is significantly smaller than the size of the stream, and the size of a query is constant (then σ is constant) the results become $S_{\texttt{net}} = O(s)$ and $T_{\texttt{net}} = O(s)$.

VI. Experimental Results

For experimentally evaluating the performance of the SPEX model, a (rather straightforward) prototype has been implemented in Java and tested against databases of various sizes and characteristics: The Mondial geographical database [12] as a small and highly structured XML document, an excerpt of the lexical WordNet database [13] as a medium sized, flat, and highly repetitive RDF representation, and the structure and content of the Open Directory Project (DMOZ) [14] as large, flat RDF documents. Four classes of queries have been considered for comparing the efficiency of the SPEX prototype with existing regular path expressions implementations:

1. simple structural queries that do not create nested results, e.g. `_*.province.city`, and `_*.Noun.wordForm`, and `_*.Topic.Title`.
2. queries with structural qualifiers that create "future conditions", i.e. conditions on candidate answers whose values are unknown at the time the candidate answers are encountered, e.g. `_*.country[province].name`, and `_*.Noun[wordForm]`, and `_*.Topic[editor].Title`.
3. structural queries that create nested results, e.g. `_*._`.
4. queries with structural qualifiers that create "past conditions", i.e. conditions on candidate answers whose values are already known when the candidate answers are encountered, e.g. `_*.country[province].religions` and `_*.Topic[editor].newsGroup`.

Fig. 14 Comparison between processors for small and medium-size documents

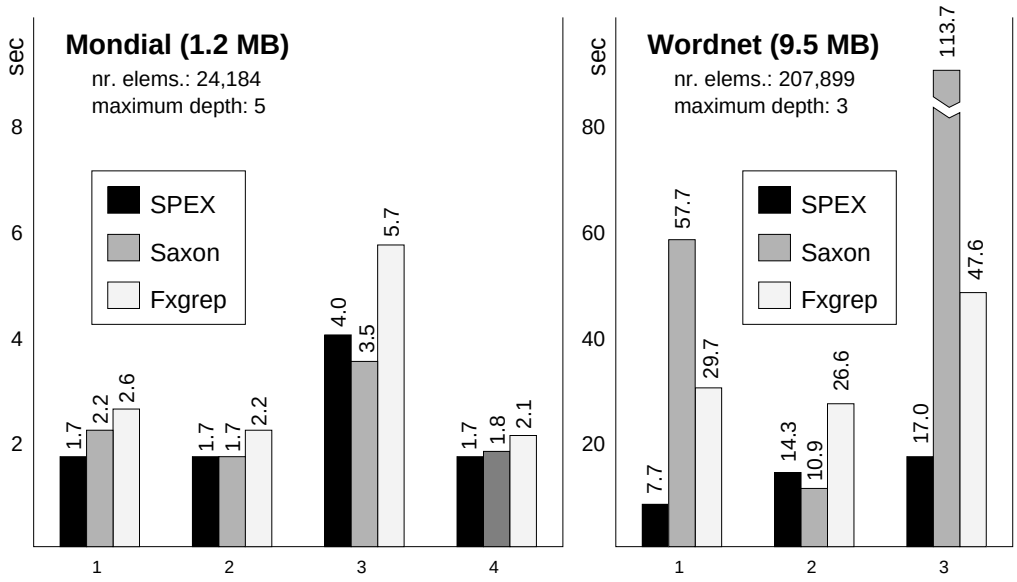


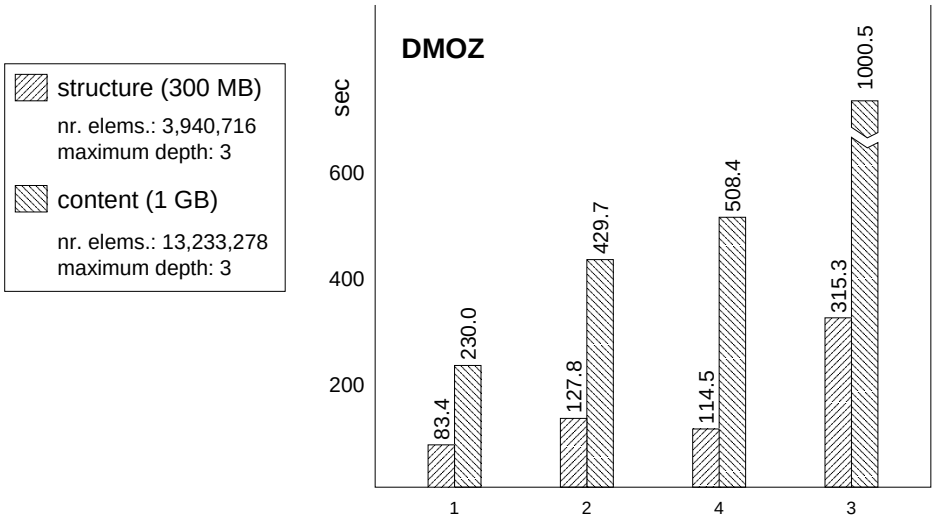
Figure 14 shows the results of processing queries from the above-mentioned classes on the MONDIAL and WordNet databases using a Pentium III 1 GHz, 512 MB system running under SuSE Linux 7.3. Processors used for comparison are the Saxon XSLT processor [15] and Fxgrep [16], an evaluator for regular tree expressions. The times given in Figure 14 for SPEX includes the compilation of the rpeq into a SPEX network.

These results show that the SPEX prototype achieves a very competitive performance on the smaller MONDIAL database and in most cases outperforms the other processors on the medium-sized WordNet database. A further comparison of the processors on larger databases like the DMOZ files could not be performed, for the memory consumption of both Saxon and Fxgrep was beyond the limitations of the system used. In contrast, the SPEX prototype, uses a constant amount of memory (between 8.5 and 11 MB, including the Java Virtual Machine) for all of the given queries and documents. The results of the query processing for the DMOZ documents are given in Figure 15, demonstrating that the proposed evaluation model can also be used for querying very large documents.

VII. Migrating to Conjunctive Queries with Regular Path Expressions

In the following, it is sketched how the SPEX model can be enhanced so as to accommodate conjunctive regular path queries with variables. Such an extension is a first step towards a streamed and progressive evaluation of query languages such as XPath and XQuery.

Fig. 15 Query processing for large and very large-size documents using SPEX networks



Definition 4. A conjunctive query is an expression of the form

$$CQ : q(\overline{X}) : -Y_1 r_1 Z_1, \ldots, Y_n r_n Z_n, \quad n \geq 1.$$

where $var(CQ) = \{Y_1, \ldots, Y_n, Z_1, \ldots, Z_n\}$ is the set of query variables of CQ, $r_1, \ldots, r_n$ are regular path expressions, and $\overline{X} \subseteq var(CQ)$ denotes the head variables of CQ. (The Y_i $(1 \leq i \leq n)$ are not necessarily pairwise distinct.)

For example, the conjunctive query $q(X_3) : -Root(\_^*.a)X_1, X_1(b)X_2, X_1(c)X_3$ selects `c` labeled nodes that are descendants of `a` labeled nodes having `b` labeled children. *Root* is a special variable always bound to the document root. Note that this conjunctive query is equivalent to the *rpeq* query from Section III.10. For integrating conjunctive queries in the SPEX framework, the following changes are necessary. A SPEX network corresponding to a conjunctive query has a sink for each head variable of the conjunctive query. A path in a conjunctive query that does not lead to a head variable corresponds to a qualifier. The translation of a conjunctive query into a SPEX network is sketched in Figure 16 in terms of a function $\mathcal{T}$ mapping a network σ and an environment *env* to an updated network and environment. There, an environment specifies bindings of variables to (transducer) tapes. The function $reach(Z, \overline{X})$ checks whether Z is on a path leading to a head variable, and function $head(Z)$ checks whether Z is a head variable. Some issues are left out, e.g. identity-based joins expressed in conjunctive queries by variables reachable via distinct paths. These issues will be the subject of future work.

Fig. 16 A denotational semantics for conjunctive queries

$$\mathcal{T} : \text{CQ} \rightarrow (\text{Network}, \text{Environment}) \rightarrow (\text{Network}, \text{Environment})$$

$$\mathcal{T}[\![Y_1 R_1 Z_1, \ldots, Y_n R_n Z_n]\!] \quad (\sigma, env) = \mathcal{T}[\![Y_n R_n Z_n]\!](\ldots(\mathcal{T}[\![Y_1 R_1 Z_1]\!](\{\mathsf{IN}_t^{in}\}, (Root, t) :: nil))\ldots)$$

$$\mathcal{T}[\![Y\ R\ Z]\!](\sigma, env) \quad = \texttt{if}\ (reach(Z, \overline{X}))$$

$$\texttt{if}\ (head(Z)) \quad ((\{\sigma_1, \mathsf{OU}_Z^{t_1}\}, env) \mid (\sigma_1, t_1) = \mathcal{C}[\![R]\!](\sigma, getTape(Y, env)))$$

$$\texttt{else} \quad ((\sigma_1, (Z, t_1) :: env) \mid (\sigma_1, t_1) = \mathcal{C}[\![R]\!](\sigma, getTape(Y, env)))$$

$$\texttt{else} \quad ((\sigma_1, (Z, t_1) :: env) \mid (\sigma_1, t_1) = \mathcal{C}[\![\ [R]\ ]\!](\sigma, getTape(Y, env)))$$

VIII. Related Work

There are by now several proposals towards an efficient streamed evaluation of XML queries [2], [17], [18], [19]. SPEX adds to this common effort a formal framework and reasonable features, by keeping in the same time the expressiveness of the supported XML query language (at least) as powerful as of the above-cited proposals. To the best of our knowledge, SPEX is the only current approach that accomodates XPath qualifiers, closure and backward navigation.

The query operator X-Scan from the Tukwila data integration system [2] compiles regular path expressions into deterministic finite automata (DFA). The DFA reports to a host application when a node is reached in a final state. An improved version [18] considers an evaluation model based on the on-demand (lazy) creation of DFA. An interesting result of [18] is its scalability to tens of thousands of queries. Both approaches use also stacks for keeping track of previous states. In [18] some expressions can be considered qualifiers, but their relations to the other expressions are left to a host application. SPEX is based on the evaluation of these connections between expressions inside and outside qualifiers and does not need extra logic for providing correct result.

The XFilter [17] and YFilter [19] engines are used for deciding if entire XML documents are matched by XPath expressions that represent user profiles. Therefore, they are not focused on answering XPath expressions. YFilter [19] proposes also a basic multi-query optimization technique for reusing common prefixes of several queries.

XSM [20] was developed in parallel to SPEX. Although both use a novel approach for a query execution plan based on transducer networks, their evaluation models, transducer types and query language features are quite different. SPEX is designed for low computational power and memory usage, which we consider essential in a stream-based context. It uses strongly

coupled (i.e. without in-between queues) pushdown transducers and supports closure steps and qualifiers. XSM uses more general loosely coupled transducers with unbounded buffers and (therefore) supports value-based joins and element creation constructs.

A problem closely related to a streamed and progressive evaluation of regular path expressions with qualifiers is the validation of XML streams under memory constraints. In [21] DTD validation and strong validation, i.e. checking the well-formedness of XML documents, are investigated. It is shown that for certain DTD classes, the validation can be done by an FSA. However, in general, the validation requires the computational power of PDA, where the size of the stack is bounded in the depth of the XML document.

IX. Conclusion

In this paper, the SPEX model for a progressive evaluation of regular path expressions with qualifiers against XML data streams has been described. With this model, a regular path expression is translated into a network of pushdown transducers. The approach needs less memory than standard approaches that store the document tree in memory. This is beneficial e.g. for mobile devices with limited memory, for data-centric applications that handle large amounts of data, for continuous services, and for a selective dissemination of information (SDI).

A salient feature of the SPEX model is the use of communicating pushdown transducers. The SPEX model requires no more computational power than necessary for querying well-formed XML documents, i.e. it is within the 1-DPDT class. Furthermore, the memory space needed by a SPEX transducer network is at most quadratic in the depth of the document tree in encountered practical cases. The size of the SPEX transducer network into which a regular path expression with qualifiers is translated is linear in the size of the regular path expression. Experiments with a prototype demonstrate a remarkable efficiency: Applied on different kinds of databases, a prototype implementation of the SPEX approach mostly outperformed standard regular path expression processors used for comparison.

Issues for further research are the migration of SPEX to conjunctive queries with variables and the integration of multi-query optimization techniques in the evaluation model. A single transducer network can be used for processing several queries having common subparts. Such a multi-query processor could be a corner stone of efficient XSLT and XQuery implementations.

References

[1] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions," in *Proc. 18th Int. Conf. on Data Engineering*, 2002.

[2] A. Levy, Z. Ives, and D. Weld, "Efficient Evaluation of Regular Path Expressions on Streaming XML Data," Tech. Rep., Univ. of Washington, 2000.

[3] T. J. Green, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata and Stream Indexes," Tech. Rep., Univ. of Washington, 2001.

[4] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web. From Relations to Semistructured Data and XML*, Morgan Kaufmann, 2000.

[5] W3C, "XML Path Language (XPath) Version 1.0," W3C Recommendation, 1999, http://www.w3.org/TR/xpath.

[6] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams: A new class of data management applications," Tech. Rep. CS-02-04, Brown Computer Science, February 2002.

[7] D. Olteanu, T. Kiesling, and F. Bry, "An Evaluation of Regular Path Expressions with Qualifiers against XML Streams," Tech. Rep. PMS-FB-2002-12, Univ. of Munich, 2002.

[8] D. Megginson, "SAX: The Simple API for XML," 1998.

[9] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, 1972.

[10] D. Olteanu, H. Meuss, T. Furche, and F. Bry, "XPath: Looking Forward," in *Workshop on XML-Based Data Management*, 2002, Springer LNCS 2490.

[11] D. Cohen, *Introduction to Computer Theory*, John Wiley & Sons, 1991.

[12] W. May, "Information extraction and integration with Florid: The Mondial case study," Tech. Rep. 131, Univ. of Freiburg, 1999.

[13] C. Fellbaum, Ed., *WordNet – An Electronic Lexical Database*, MIT Press, 1998, available at `http://www.cogsci.princeton.edu/~wn/`.

[14] "dmoz – The Open Directory Project," located at `http://dmoz.org/`.

[15] "Saxon 6.5.2," available at `http://saxon.sourceforge.net/`.

[16] A. Neumann, "Fxgrep: The functional XML querying tool," 2000.

[17] M. Altinel and M. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," in *Proc. 26th Int. Conf. on Very Large Data Bases*, 2000.

[18] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata," Tech. Rep., Univ. of Washington, 2002.

[19] Y. Diao, P. Fischer, M. J. Franklin, and R. To, "YFilter: Efficient and Scalable Filtering of XML Documents," in *Proc. 18th Int. Conf. on Data Engineering*, 2002.

[20] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou, "A Transducer-Based XML Query Processor," in *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002.

[21] L. Segoufin and V. Vianu, "Validating Streaming XML Documents," in *Proc. 21st ACM Symp. on Principles of Database Systems*, 2002.