The XML Stream Query Processor SPEX

François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, Markus Spannagel
Institute for Informatics, University of Munich, Germany
{bry,furche,olteanu}@pms.ifi.lmu.de {coskun,durmaz,spannage}@stud.ifi.lmu.de

1. Introduction

Data streams (e.g., [1]) 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. Areas where data streams are applied include monitoring of scientific data (astronomy, meteorology), control data (traffic, logistics, networks), and financial data (bank transactions). Data streams are a new and promising setting in which many conventional database methods have to be considered anew. Querying XML data streams without storing and without decreasing considerably the data throughput is especially challenging because XML streams convey tree structured data with (possibly) unbounded size and depth.

SPEX, initially described in [3], evaluates XPath queries against XML data streams. SPEX is built upon formal frameworks for (1) rewriting XPath queries into equivalent XPath queries without reverse axes [4] and (2) correct query evaluation with polynomial combined complexity using networks of pushdown transducers [2]. Such transducers are simple, independent, and can be connected in a flexible manner, thus allowing not only easy extensions but also extensive query optimization, e.g., by sharing transducers. A reason for the latter is that processing new query constructs implemented by new transducers does not affect the processing of existing ones. As a proof of concept, SPEX is extended here with novel compile-time optimizations that reduce both the size of the transducer network and the processing of irrelevant stream fragments.

SPEX is demonstrated using a practically useful application for monitoring processes running on UNIX systems, and a novel, sophisticated visualization of its run-time system, called SPEX Viewer. SPEX Viewer makes it possible to visualize (1) the step-by-step rewriting of XPath queries into equivalent queries without reverse axes, (2) the networks of pushdown transducers generated from such queries, (3) the incremental processing of XML streams with these networks under various novel optimization settings, and (4) the progressive generation of answers.

2. Application scenario: Monitoring Computer Processes

For demonstrating the SPEX query processor, a concrete application is used: monitoring processes currently running on UNIX computers. The process parameters are constantly gathered as a continuous XML stream from the output of the `ps -elfH` command.

The XML stream generated in this manner is unbounded in size and depth, because (1) new process information wrapped in XML is repeatedly sent in the stream and (2) the process hierarchy can contain arbitrarily nested processes.

By means of XPath queries the monitoring application allows the user to specify what process information is to be watched and reported back. One can, e.g., monitor suspended processes with CPU and memory expensive subprocesses. Monitoring queries can also express simple aggregations, e.g., the selection of processes that together with their subprocesses use a certain amount of memory or have more than a given number of subprocesses.

The combination of the XML encoding of process information used here and an XML stream query evaluator like SPEX turns out to be a natural, declarative, and effective solution for monitoring relations between nested processes.

3. The SPEX Query Processor

Querying XML streams with SPEX consists in four steps: First, the input XPath query is rewritten into an XPath query without reverse axes [4]. Second, the forward XPath query is compiled into a logical query plan abstracting out details of the concrete XPath syntax. Then, a physical query plan is generated by extending the logical query plan with operators for determination and collection of answers. In the last step, the XML stream is processed continuously with the physical query plan, and the output stream conveying the answers to the original query is generated progressively. All four steps are further detailed below.

Step 1: Source-to-source query transformations. The forward and reverse XPath axes enable random access to nodes of an XML tree. If queries are to be evaluated against streams conveying XML trees, nodes cannot be accessed randomly, but rather in the stream's sequence. The evaluation of reverse axes, e.g., ancestor and preceding, would demand then the (possibly unnecessary) buffering of already processed stream fragments. SPEX proposes a framework [4] for rewriting queries with reverse axes into equivalent queries in which only forward axes occur.

Further source-to-source transformations that optimize the evaluation of forward XPath queries are also applied in this step. Such optimizations focus on pruning redundant computations. E.g., consider the query /child::process/following::state that selects all state-elements following process children of the root. For the set of state-elements that follow the first process child of the root is already the set of state-elements that follow all process children of the root, this query can be rewritten to /child::process[1]/following::state, so that only the first process child of the root is considered during evaluation.

Step 2: Compilation into a logical query plan. A forward XPath query is compiled into a logical query plan that consists either in a path, if the query is a sequence of steps, in a tree, if the query also has predicates, or in a directed acyclic graph, if the query also has set operators. Each construct in a forward XPath query, such as an axis or a predicate, induces a corresponding operator in the logical query plan.

At this step, further compile-time optimizations can be applied: E.g., common prefixes of branches rooted at an and (or) operator are compacted. Note that such a "branch compaction" is not possible at the level of the XPath syntax and therefore not possible at the first step.

Step 3: Generation of a physical query plan. A physical query plan is a transducer network that computes the answers to the initial query from the XML stream. Such a network is created from a logical query plan in two steps.

First, each operator from a logical query plan is realized in a network as a deterministic pushdown transducer.

Second, the network is extended at its beginning with a stream-delivering transducer in, and at its end with an answer-collecting *funnel*, i.e., a subnetwork of auxiliary transducers serving to collect the computed potential answers. For each and, or, or not predicate in the query there is a pair of transducers in the network implementing the logic of that predicate. The nesting of such pairs corresponds to the nesting of predicates in the query. An answer transducer is introduced to annotate potential answers that are buffered by the out transducer. The out transducer also delivers the query answers.

Step 4: Processing with a physical query plan. Processing an XML stream corresponds to a depth-first left-to-right preorder traversal of the (implicit) XML tree conveyed by that stream. Exploiting the affinity between preorder traversal and stack management, the transducers use their stacks for remembering the depth of the nodes in the implicit XML tree. This way, binary relations expressed as forward XPath axes, e.g., child and desc, can be computed in a single pass. A transducer network processes the XML stream annotated by its first transducer in. The other transducers in the network process stepwise the received annotated XML stream and send it with changed annotations to their successor transducers. E.g., a transducer child moves the annotation of each node to all children of that node.

The answers computed by a transducer network are among the nodes annotated by the answer transducer. These nodes are *potential* answers, as they may depend on a downstream satisfaction of predicates. The information on predicate satisfaction is also conveyed in the stream by annotations. Until the predicate satisfaction is decided, the potential answers are buffered by the out transducer.

Those optimizations that are specific to stream processing are applied only to the transducer network. Specialized transducers, called structural filters, are employed to minimize the stream traffic between transducers in a network. They can depend both on the structure of the transducer network and of the data.

4. SPEX Viewer

The SPEX processor is visually demonstrated using the SPEX Viewer. It illustrates the four steps of the SPEX processor, in particular showing the stepwise query rewriting and stream processing together with the progressive generation of answers, and also windows over the most recent messages from the input XML stream and the most recent answers. The SPEX Viewer provides three processing modes: step-by-step, running, and pause mode. Breakpoints can be specified to alert when a given XML tag reaches given transducers, or when given transducers have particular stack configurations.

The SPEX Viewer can give a concrete feeling for the polynomial combined complexity of SPEX [2] and for the influence of various optimizations on the stream processing.

References

[1] N. Koudas and D. Srivastava. Data stream query processing: A tutorial. In *Proc. of VLDB*, 2003.

[2] D. Olteanu, T. Furche, and F. Bry. Evaluating Complex Queries against XML streams with Polynomial Combined Complexity. In *Proc. of BNCOD*, 2004.

[3] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *Proc. of ICDE*, 2003.

[4] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. of EDBT Workshops*, 2002. LNCS 2490.