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). They are complementary and symmetrical to traditional databases: While in traditional databases data is persistent and queries are volatile, in data stream applications data is volatile but queries are persistent. 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. XPath is a key query language used, e.g., in standards like XQuery, XSLT, XML-Schema, XLink, XPointer etc., for expressing selection of or reference to XML data. 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 complexity using networks of pushdown transducers [2]. Such transducers are simple, independent, and can be connected in a flexible manner. They communicate through annotations added to the input stream. SPEX allows easily not only extensions (for processing new query constructs implemented by new transducers does not affect the processing of existing ones) but also extensive query optimization (e.g., by sharing transducers). 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. The monitoring application demonstrates well the features of SPEX, i.e., (1) the processing of XML streams with recursive structure definition and unbounded size as gathered from the information about UNIX processes, and (2) the detection of specific patterns in such richly structured streams based on the evaluation of rather complicated XPath queries. SPEX Viewer makes it possible to visualize (1) the 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 information about a process is represented as an XML element process containing child elements for various properties of a process, such as memory and time used, current priority and state, and child processes. Thus, the process hierarchy is represented by ancestor-descendant relations between process-elements.

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[1].

By means of XPath queries (for the sake of simplicity but without loss of generality, omitting value-based comparisons of subquery results and providing restricted support for positional predicates) the monitoring application allows the user to specify what process information conveyed in the XML stream is to be watched and reported back. One can, e.g., monitor suspended processes with CPU and memory expensive subprocesses. More specifically, these can be

[1]In practice, many Linux versions allow at most 512 processes running at a time on one machine, thus limiting the process hierarchy depth to 512.

```
/desc::process[child::time > 24 or child::memory > 500]/anc::process[child::priority < 10 and child::state = ”stopped”]
```

Figure 1. Sample query for monitoring processes (desc **and** anc **abbreviate** descendant **and** ancestor**)**

```
/desc::process[child::priority < 10 and child::state = ”stopped” and desc::process[child::time > 24 or child::memory > 500] ]
```

Figure 2. Equivalent forward XPath query for query of Figure 1

processes with a certain low priority (e.g., below 10) that are currently stopped and are ancestors of at least one process in the process hierarchy. Furthermore, this other process must use more than 500 MB main memory or be already running for more than 24 hours. The corresponding XPath query is shown in Figure 1.

Monitoring queries can also express simple aggregations, e.g., so as to select processes that together with their subprocesses use a certain amount of memory or that have more than a given number of subprocesses. Note that rather complex and possibly nested queries can be expressed in XPath and processed with SPEX. Query nestings reflect process nestings expressed in the XML stream. 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, as shown in Figure 3. First, the input XPath query is rewritten into a forward XPath query [4], i.e., without reverse axes. For the query of Figure 1, the result of this source-to-source transformation is shown in Figure 2. The forward XPath query is compiled into a logical query plan that abstracts out details of the concrete XPath syntax. Figure 4 gives a logical query plan for the query of Figure 2. Then, a physical query plan is generated by extending the logical query plan with operators for determination and collection of answers. Figure 5 shows a physical query plan for the logical query plan of Figure 4. In the last step, the XML stream, which in the chosen application scenario consists in information about the status of processes, 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 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. E.g., the query of Figure 1 is rewritten into the query of Figure 2.

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 has also predicates, or in a directed acyclic graph, if the query has also 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. Figure 4 shows the logical query plan for the query of Figure 2. Square boxes denote the answers sought for, round boxes correspond to (parts of) predicates. E.g., the and (or) operator of Figure 4 expresses that both (at least one of the) subplans rooted at the subjacent child operator further constrain the answers selected by the first process operator.

At this step, further compile-time optimizations can be applied. As shown in Figure 4, both prefixes child of the branches rooted at the and (or) operator are compacted into a single child operator. Note that such a branch compaction is not possible at the level of XPath syntax.

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. Figure 5 shows the network constructed from the logical query plan

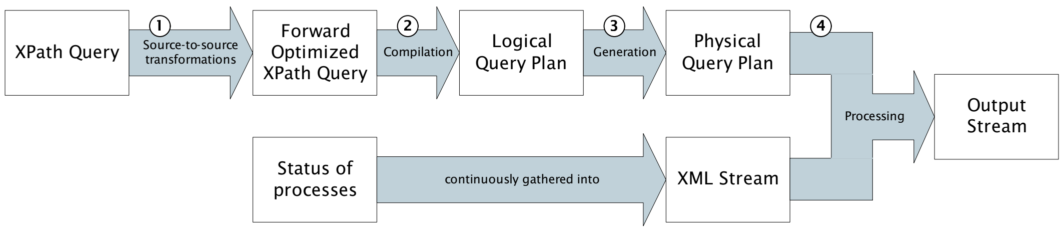


Figure 3. Steps of the SPEX processor

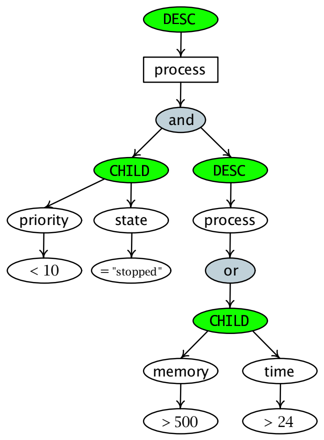


Figure 4. Logical query plan for Figure 1

of Figure 4. For each predicate in the query there is a pair (and, cd-and) or (or, cd-or) of transducers in the network (cd stands for condition determinant). The nesting of such pairs corresponds to the nesting of predicates in the query. The topmost process transducer is the answer transducer, as indicated by the square box. The last transducer of the funnel is the out transducer that buffers potential answers and 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, forward XPath axes, e.g., child and desc, can be evaluated in a single pass. A physical query plan, i.e., 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 down-

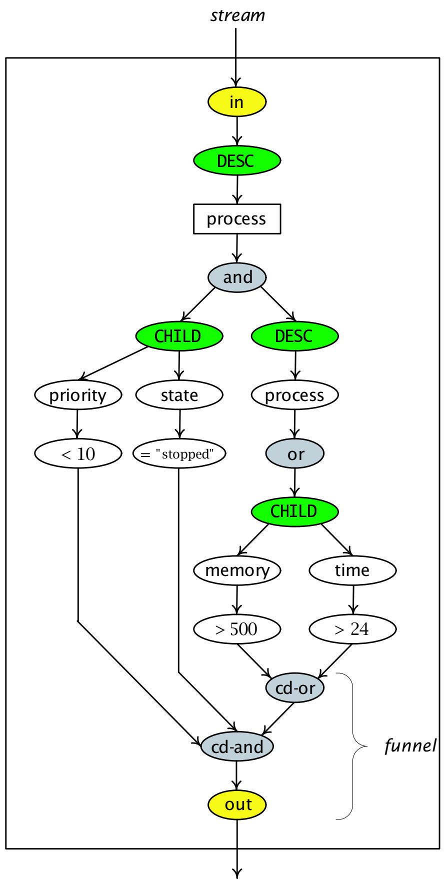
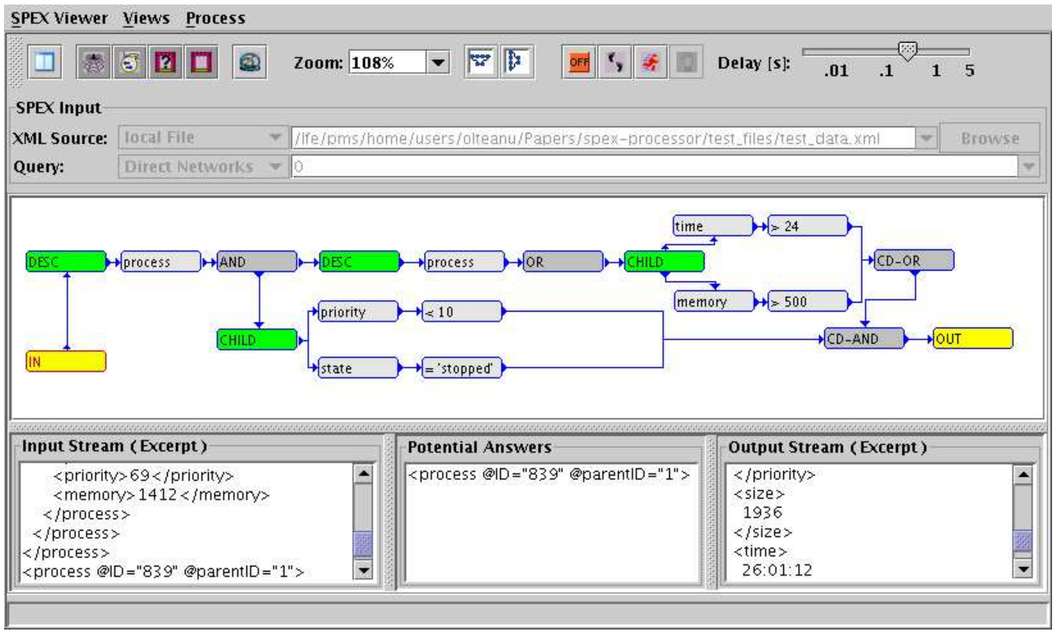


Figure 5. Physical query plan for Figure 2

stream satisfaction of predicates. The information on predicate satisfaction is conveyed in network also 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 physical query plan. Specialized transducers are employed to minimize the stream fragment processed by transducers in a network. E.g., in the physical query plan of Figure 5, all transducers after the answer transducer require only the stream fragments conveying the subtrees rooted at nodes selected by desc::process and the



Zoom: 108%

XML Source: local File /ife/pms/home/users/olteanu/Papers/spex-processor/test_files/test_data.xml

Query: Direct Networks 0

Input Stream (Excerpt)

```
    <priority>69</priority>
    <memory>1412</memory>
  </process>
 </process>
</process>
<process @ID="839" @parentID="1">
```

Potential Answers

```
<process @ID="839" @parentID="1">
```

Output Stream (Excerpt)

```
</priority>
<size>
 1936
</size>
<time>
 26:01:12
```

Figure 6. SPEX Viewer illustrates how SPEX processes XML streams

irrelevant stream fragments can be filtered out by an appropriate pushdown transducer placed after the answer transducer. Various structural filters can be added to physical query plans, depending on the kind of transducers existent in a network and on the stream structure. The latter dependencies can be derived, e.g., from schemas of the stream.

4 SPEX Viewer

The SPEX processor is demonstrated using the SPEX Viewer, that visualizes how SPEX processes XML streams. The salient features of the SPEX Viewer consist in illustrating the four steps of the SPEX processor, in particular showing (1) the logical and physical query plans, (2) the stepwise processing of XML streams with physical query plans together with the progressive generation of answers, and (3) windows over the most recent messages from the input XML stream and the most recent answers.

A vector-based graph rendering engine has been designed and implemented that fits the needs of demonstrating SPEX. Since query plans and SPEX transducer networks may be quite large, reversible actions like moving, hiding parts, and zooming are offered. As displayed transducer stacks change during query processing in content and size, automatic on-line graph reshaping is provided. Figure 6 shows a rendering of the physical query plan of Figure 5 in the middle area of the visualization tool. The lower area shows (from left to right) windows over the most recent fragment of the input XML stream, over the current potential answers, and over the most recent query answers.

For a detailed insight into the processing, three processing modes are provided that can be switched at any time during processing. In the *step-by-step mode*, the content of each transducer stack and the message passing between transducers can be inspected for each incoming stream message. In the *running mode*, the input stream is processed message after message with a speed chosen by the user (cf. the delay slider on the topright of Figure 6). The *pause mode* is used to interrupt the processing for a detailed inspection of transducers in the SPEX transducer network. While in the pause mode, processing can be resumed by selecting either the step-by-step or the running 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 also 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 Workshop XMLDM*, 2002. LNCS 2490.