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

Ludwig
Maximilians–
Universität
München

An Efficient Single-Pass Query Evaluator for XML Data Streams

Dan Olteanu, Tim Furche, François Bry

Technical Report, Institute for Computer Science, Munich, Germany
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-2004-1, 2004

An Efficient Single-Pass Query Evaluator for XML Data Streams

Dan Olteanu, Tim Furche, François Bry
Institute for Computer Science, University of Munich, Germany
{olteanu, timfu, bry}@informatik.uni-muenchen.de

ABSTRACT

Data streams might be preferable to data stored in memory in contexts where the data is too large or volatile, or a standard approach to data processing based on data parsing and/or storing is too time or space consuming. Emerging applications such as publish-subscribe systems, data monitoring in sensor networks [6], financial and traffic monitoring, and routing of MPEG-7 [7] call for querying data streams. In many such applications, XML streams are arguably more appropriate than flat data streams, for XML data is record-like, though not precluding multiple occurrences of fields with the same name. Evaluating selection queries against XML streams is especially challenging because XML data is structured (like records) and might have unbounded size.

This paper proposes an efficient single-pass evaluator of XPath queries against XML data streams unbounded (possibly infinite) in size. The evaluator is based on networks of independent deterministic pushdown transducers and it is especially suitable for implementation on devices with low-memory and simple logic as used, e.g., in mobile computing.

Keywords

XML streams, XPath, Single-Pass Query Evaluation

1. INTRODUCTION

XML Streams are unparsed XML documents, i.e., XML documents in linear form as generated, e.g., by a Web page editor or exchanged by applications on the Internet. Well-formed XML streams convey tree-shaped data items called *XML document trees*. A t-labeled node in such a tree corresponds to a well-formed XML stream fragment called *element* beginning with an opening tag ⟨t⟩ and ending with the corresponding closing tag ⟨/t⟩. In this paper, it is assumed that the XML document trees conveyed by XML streams are not materialized. Recall that a sequential traversal of an XML stream amounts to a depth-first left-to-right preorder traversal of the XML document tree associated with the XML stream. In the following, XML streams are assumed to be well-formed. This condition might be ensured by the applications generating the XML streams. Note that the query evaluator described below can be extended so as to detect whether an XML stream is not well-formed.

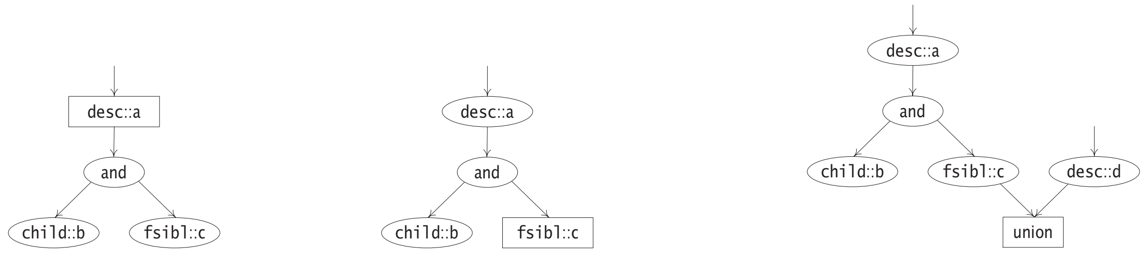
SXP. XPath has established itself as the prime language for expressing selection queries on XML documents and it is a central component of the standard Web query and transformation languages XQuery and XSLT. In the following, familiarity with XPath is assumed. For the sake of simplicity and conciseness, a core fragment of XPath referred to as "Simple XPath" (short "SXP") is considered here. SXP includes XPath's (1) forward axes child, descendant (short desc), following-sibling (short fsibl), following (short foll), self and reverse axes parent, ancestor, preceding-sibling, preceding, (2) node tests, (3) possibly nested predicates, and (4) the union, intersect, and except set operations. Among XPath features not included in SXP are the value-based comparisons of subquery results (such as [child::a = fsibl::a]) and positional predicates (such as [position() = 1]). The query evaluator described in the present paper easily extends to full-fledged XPath, though in presence of the above comparisons the efficiency results of Section 3 degrade.

XPath's (and hence SXP's) query paradigm refers to XML document trees, not to XML streams. XPath (and SXP) queries return as answers sequences of elements (XML document tree nodes). The SXP query /desc::a[child::b][fsibl::c] selects from an XML stream fragments conveying a-labeled elements that have both, b-labeled child elements and c-labeled sibling elements that follow them. The SXP query /desc::a[child::b]/fsibl::c selects fragments conveying c-labeled elements that are preceded by a-labeled sibling elements having themselves b-labeled child elements.

FSXP. The forward and reverse axes of XPath and SXP enable random access to all nodes of an XML document tree. If queries are to be evaluated against XML streams, data cannot be accessed randomly, but rather in the stream's sequence, thus making XPath's reverse axes undesirable.

Methods such as [10] have been developed for rewriting queries (within an XPath fragment semantically equivalent to SXP) including reverse axes into queries in which only forward axes occur. These methods are practicable because (1) they preserve query equivalence (i.e., both the initial and rewritten queries yield the same answers when applied to the same XML document), and (2) the rewritten query has a size linear in the size of the initial query. Let FSXP denote the fragment of SXP including only forward axes. FSXP is semantically equivalent to SXP. In the following, only the evaluation of FSXP queries is considered.

SAC'04, March 14-17, 2004, Nicosia, Cyprus



(a) /desc::a[child::b][fsibl::c] (b) /desc::a[child::b]/fsibl::c (c) /desc::a[child::b]/fsibl::c union /desc::d

Figure 1: FSXP queries and their corresponding AXP queries

2. THE QUERY EVALUATOR

The evaluator described here processes an FSXP query against an XML stream as follows. First, it generates a network of transducers from the FSXP query. Second, this network of transducers computes the answers to the initial FSXP query from the XML stream. Most transducers considered are single-state pushdown automata with output tape. The transducer networks are directed acyclic graphs.

Compiling FSXP Queries into Transducer Networks. The transducer network associated with an FSXP query is constructed in two steps.

First, the FSXP query is parsed and translated into a query in abstract syntax, which consists either in a path if the query is a sequence of axes and node tests, or in a tree if the query has also predicates, or in a directed acyclic graph if the query contains also set operators. Let AXP denote the language of queries in abstract syntax into which FSXP queries are translated. Each node in an AXP query specifies a transducer. There are transducers for each pair (axis, node test), for predicates, and for set operations.

Second, the transducer network specified by an AXP query is extended at its beginning with a stream-delivering in transducer, and at its end with an answer-collecting *funnel*, i.e., a network of auxiliary transducers serving to collect the potential answers computed by the transducer network.

The first step is illustrated by FSXP queries and their translation into AXP shown in Figure 1. Square boxes denote the answers sought for, round boxes correspond to (parts of) predicates. The and transducer expresses that both sub-networks child::b and fsibl::c specify conditions on those elements satisfying desc::a. An and transducer has two or more outgoing edges. Note that while FSXP (like XPath) requires to mark predicates (with square brackets), AXP marks answer nodes (with square boxes). Answer nodes of AXP queries are called *head* nodes or *head* transducers. It is worth pointing out that AXP can be seen as a generalization of FSXP in two ways: (1) AXP abstracts out some irrelevant (syntactical) aspects of FSXP like any abstract language used in compiling, and (2) AXP makes queries with several answer nodes (i.e., several square boxes) possible, although such queries do not correspond to FSXP queries. Allowing several answer nodes is an important step towards a single-pass multi-query evaluator on XML streams as described in [3] and as needed, e.g., in publish-subscribe systems.

The second step is illustrated by an FSXP query and its translation into a complete transducer network shown in

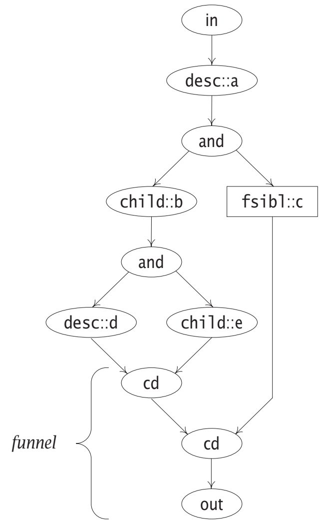


Figure 2: Transducer network for FSXP query /desc::a[child::b[desc::d]/child::e]/fsibl::c

Figure 2. For each predicate in the FSXP query there is a pair (and, cd) of transducers in the network (cd stands for condition determinant). The nesting of (and, cd) pairs corresponds to the nesting of predicates in the FSXP query. A cd transducer has as many ingoing edges as its corresponding and transducer has outgoing edges, i.e., two or more. The fsibl::c transducer is in this case the *head*, as indicated by the square box. The last transducer of the *funnel* is the out transducer that buffers potential answers and outputs the answers selected by the *head*, as explained below.

Transducer Communication. A transducer network processes the XML stream delivered by its first transducer in. Each transducer in the network processes stepwise the XML stream it receives and transmits it unchanged or annotated with conditions to its successor transducers. An annotation immediately follows an opening tag of an element. Processing an XML stream this way corresponds to a depth-first traversal of the (implicit) XML document tree associated with the XML stream. Exploiting the affinity between depth-first search and stack management, the transducers use their stacks for keeping track of the depth of the elements in the XML document tree conveyed by the XML stream. This way, forward axes, e.g., child or desc, can be evaluated in a single pass.

The answers computed by a transducer network are among the elements annotated by a *head*. These elements are *potential* answers, as they may depend on a downstream satisfaction of some predicates. The predicate satisfaction is conveyed in the transducer network by conditions with which elements are annotated. Until the predicate satisfaction is decided, the potential answers are buffered by the out transducer. Consider the evaluation of the FSXP query /desc::a[child::b]. When encountering on the XML stream an opening tag $\langle \mathsf{a} \rangle$ marking the beginning of an a-labeled element, it is not yet known whether this a-labeled element has a b-labeled child element, i.e., whether it is an answer or not. Indeed, by definition of XML streams, such a child can only appear downstream. This might remain unknown until the corresponding closing tag $\langle /\mathsf{a} \rangle$ is processed. At this point, it is impossible for the a-labeled element to have further b-labeled children because the XML stream is assumed to be well-formed. Thus, the stream fragment corresponding to a potential answer has to be buffered as long as it is not known whether predicates that might apply are satisfied or not, but no longer.

Conditions are generated as follows. The in transducer (cf. Figure 2) annotates the first element in the XML stream with an initial condition [1], which is considered satisfied. Each of the next elements is annotated with the empty condition [ ], which is always unsatisfied. Upon receiving an element e annotated with a condition $[c]$ (different from [ ]), generated by the in transducer or a previous and transducer, an and transducer (that expresses FSXP predicates, cf. Figures 1 and 2) replaces this condition with a novel condition $[n]$. n is the *running condition number* of the and transducer, which is its stack size. The and transducer increases its stack size by 1, and forwards the condition mapping $[c] \rightarrow [n]$ to its corresponding cd transducer and the condition $[n]$ to all its outgoing edges. When $[n]$ is collected from all ingoing edges of the cd transducer, $[n]$ is considered satisfied. Using the received condition mapping $[c] \rightarrow [n]$ the cd transducer converts $[n]$ back to $[c]$ and forwards $[c]$ to an ingoing edge of the cd transducer corresponding to the preceding and transducer that created $[c]$, or of the out transducer corresponding to the in transducer. However, as soon as it is known that $[n]$ can no longer be satisfied, $[n]$ is considered unsatisfied. E.g., if the second and transducer in Figure 2 receives the closing tag of the element e (that was annotated with $[c]$), then its stack size is decreased by 1. When the same closing tag reaches also the corresponding cd transducer and $[n]$ was not yet satisfied, then $[n]$ is considered unsatisfied. In this case, the elements annotated with $[n]$ by a *head* and buffered by the out transducer are discarded.

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

Specification of Transducers for Forward Axes. All transducers (except the auxiliary cd and out transducers) are single-state deterministic pushdown transducers $(q, \Sigma, \Gamma, \delta)$, where q is the single state; Σ the input and output alphabet consisting of all opening and closing tags, e.g., $\langle \mathsf{t} \rangle$ and $\langle /\mathsf{t} \rangle$, and conditions; Γ is the stack alphabet consisting of all conditions; the transition function δ is canonically extended to the configuration-based transition function $\vdash : \Sigma \times \Gamma^* \rightarrow \Gamma^* \times \Sigma^*$ (used below as infix operator). In the following specifications, $[c] \mid \gamma$ reads: $[c]$ is the top of the stack $|$-separated from the rest of the stack γ. $[c]$ stands for a condition, like $[1, 2]$, or [ ]. $[c] \cup [s]$ denotes the set union of $[c]$ and $[s]$.

Configuration-based transitions defining the child::a, desc::a, fsibl::a, and foll::a transducers are given in the following. These configurations differ only in the first transition. Actually, the first transition of each transducer definition is a compaction of several simpler transitions that do only one stack operation. Note that the node test a is just a parameter, and it can be replaced by any other node test, including the wildcard (matching any tag label). The tag label x stands for any tag label but a. In the case of a wildcard node test, the transitions 4 and 5 can be dropped (cf. child::a transitions below), for the differentiate treatment for matching vs. non-matching labels is not needed.

Every element, say e, received by a transducer is annotated with a condition $[c]$. Recall that an element is annotated with a condition when that condition follows immediately the element opening tag in the XML stream. An r::a transducer for an axis r and a node test a identifies each element e' from the incoming stream that stands in relation r::a with e and annotates it with $[c]$. E.g., for a child::a transducer, the elements e' are a-labeled children of e.

Assume that $[s]$ is the topmost condition of the stack of the r::a transducer when it receives an element e annotated with a condition $[c]$. Depending on where $[c]$ is stored on the stack, the following transducer specifications show four possible cases corresponding to four different axes:

child::a **transducer.** If $[c]$ is pushed alone as a new stack entry, then each a-labeled element e' child of e is annotated with $[c]$. The next message received can be (1) a new opening tag, corresponding to an a-labeled element e' child of e followed by a condition $[c']$, (2) a new opening tag corresponding to an x-labeled element e'', or (3) the closing tag of e. In the first case, the topmost condition (now $[c]$) is output together with the opening tag of e' and $[c']$ is pushed as a new stack entry. When the closing tag of e' is received, $[c']$ is popped from the stack and new a-labeled elements that are children of e can be annotated with the topmost stack entry $[c]$. In the second case, the label x of the element e'' is not matched by a and therefore e'' is annotated with the empty condition [ ], and not with $[c]$. In the third case, $[c]$ is popped from the stack, for there are no children of e left.

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

desc::a **transducer.** If $[c]$ is stored together with the previous top $[s]$ as a new stack entry, then each a-labeled element e' descendant of e is annotated with $[c]$. Indeed, all a-labeled descendants e' are annotated with $[c]$ because

(1) the previous top ($[s]$) is always carried in the current top ($[c]\cup[s]$), and (2) the incoming a-labeled elements are always annotated with the topmost condition.

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

The transitions (2 to 5) are as for the child::a transducer.

fsibl::a transducer. If $[c]$ is stored together with the previous top $[s]$, and an empty condition $[\,]$ becomes the current top, then each sibling a-labeled element that follows e is annotated with $[c]$. On receiving the closing tag of e, $[c]\cup[s]$ becomes the topmost condition on the stack. From now on, each a-labeled element e' sibling of e is annotated also with $[c]$. When the closing tag of the parent of the element e is read, then also $[c]\cup[s]$ is popped, for there are no siblings of e left.

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

The transitions (2 to 5) are as for the child::a transducer.

foll::a transducer. If $[c]$ is stored together with the previous top $[s]$, and $[s]$ becomes also the current top, then each a-labeled element that follows e is annotated with $[c]$.

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

The transitions (2 to 5) are as for the child::a transducer.

The first transitions of the above transducers can be compared as follows: if $[c]$ is pushed as a new stack entry, then child elements of e are annotated with $[c]$. Carrying on the previous top gives rise to closure relations, like desc (vertical closure) or fsibl (horizontal closure). E.g., if the previous top is also part of the current top, then descendant elements of e are annotated with $[c]$. If the previous top remains in its place, then following sibling elements of e are annotated with $[c]$. Restrictions or combinations of these behaviours can give rise to other non-trivial relations, e.g., first-child (selecting only the first child of an element), first-fsibl (selecting only the first following sibling of an element), or child-or-fsibl (selecting all children and following siblings of an element).

The self::a **transducer** performs only the associated node test a, i.e., it forwards only conditions received for a-labeled elements. For the other elements it replaces the conditions with the empty condition $[\,]$.

Set Operation Transducers. The set operation transducers are similar to the and transducer. However, as set operations are defined for $k \geq 2$ operands, the set transducers have k ingoing edges and their corresponding cd transducers have k outgoing edges, one edge for each sub-network implementing an operand. For each received element, a set transducer receives also a condition $[c_i]$ ($1 \leq i \leq k$) from each ingoing edge, and possibly maps all $[c_i]$ to a new condition $[n]$. The intersect transducer creates the new condition $[n]$ only if all $[c_i]$ conditions are non-empty, the union transducer creates $[n]$ if at least one of the received conditions is non-empty. The except transducer assumes an order between its ingoing edges: the first ingoing edge is the minuend, whereas the others are subtrahends. Hence, it creates a new condition only if the first condition $[c_1]$ is non-empty.

If a *head* has annotated potential answers with $[n]$, then they become answers only when $[n]$ is satisfied (i.e., it is received by the cd transducer), at least one condition that is mapped directly or indirectly to $[n]$ from each predicate scope is satisfied, and (1) for the union transducer at least one $[c_i]$ condition is satisfied, (2) for the intersect transducer all $[c_i]$ conditions are satisfied, (3) for the except transducer only the first condition $[c_1]$ is satisfied.

3. COMPLEXITY

Analytical Results. The FSXP query evaluator described in Section 2 has a polynomial combined complexity in both the stream and the query size, which is near the theoretical optimum for in-memory FSXP evaluation [4]. Due to space reasons, the complexity of the query evaluator for a query of size q is presented in the following without proofs. Former investigations [9] report on closely related complexity results. It is assumed that the XML document tree conveyed by the XML stream can have recursive structure definition and has maximal depth d, maximal breadth b, and size s. In most practical cases the depth d is by orders of magnitude smaller than the size s. The space S_i and time T_i complexities are given for five fragments of FSXP ($1 \leq i \leq 5$):

(1) FSXP_1 contains all axes, wildcard, and set operations, but no predicates; $S_1 = O(q \times d)$ and $T_1 = O(q \times s)$.

(2) FSXP_2 contains child and self axes, wildcard, set operations, and predicates; $S_2 = O(q \times d + s)$ and $T_2 = O(q \times s)$.

(3) FSXP_3 contains FSXP_2 and the desc axis. If in or after a predicate desc is the first axis, or all axes before the first desc have a wildcard node test, then $S_3 = S_2$, $T_3 = T_2$. Otherwise, $S_3 = O(q \times d^2 + s)$ and $T_3 = O(q \times d \times s)$.

(4) FSXP_4 contains FSXP_3 and the fsibl axis. If fsibl occurs as the first axis in or after a predicate, then $S_4 = O(q \times d \times \max(d, b) + s)$ and $T_4 = O(q \times \max(d, b) \times s)$. Otherwise, $S_4 = S_3$, $T_4 = T_3$.

(5) FSXP_5 contains FSXP_4 and the foll axis; $S_5 = O(q \times d \times s)$ and $T_5 = O(q \times s^2)$.

For evaluating queries with predicates ($i \geq 2$), the extra space s can be needed for buffering potential answers. As explained in Section 2, buffering potential answers is independent of the present query evaluator and in some cases unavoidable. The entire space s is needed for buffering only in pathological cases, e.g., when the entire XML stream is a potential answer that depends on a condition satisfaction which can be decided only at the end of the XML stream.

Among the presented FSXP fragments, only FSXP_1 is suitable for querying infinite XML streams. For the other fragments, advanced approximation techniques for query evaluation under memory constraints can be used, e.g., [12].

Experimental Results. The theoretical results have been verified by experiments with a prototype implemented in Java (Sun Hotspot JRE 1.4.1) on a Pentium 1.5 GHz with 500 MB under Linux 2.4.

XML Streams. The effect of varying the stream size s on query evaluation time is considered for two XML stream sets. The first set [8] provides real-life XML streams, ranging in size from 21 to 21 million elements and in depth from 3 to 36. The second set provides synthetic XML streams with a slightly more complex structure that allows more precise variations in the workload parameters.

Queries. Only FSXP queries that are "schema-aware" are considered, i.e., that express structures compatible with the schema of the XML streams considered. Their generation has been tuned with the query size q and several probabilities for query constructs: p_{fsibl} for fsibl, p_{desc} for desc, $p_{[\,]}$ for predicates, $p_{\{\}}$ for set operations, and p_{wild} for wildcard. E.g., a query without set operations is obtained with $p_{\{\}} = 0$, whereas a simple path query with $p_{[\,]} = p_{\{\}} = 0$. If not varied, all probabilities are set to 0.5. For each parameter setting 10–30 queries have been tested, totaling about 1200 queries.

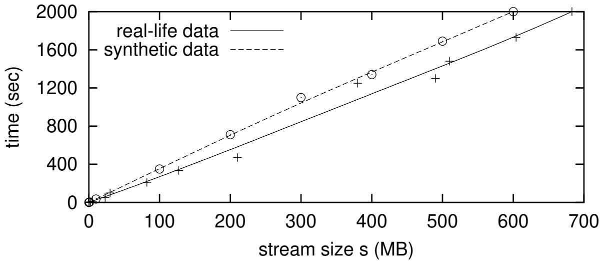


Figure 3: Varying stream size s **(**$q = 10$**,** $d \leq 32$**)**

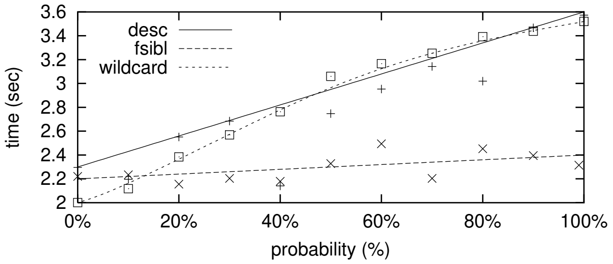


Figure 4: Effect of p_{wild}**,** p_{desc}**, and** p_{fsibl} **(**$s = 244$ **KB,** $d = 32$**,** $q = 10$**)**

Scalability. The depth d has shown to be considerably less influential on processing time than the stream size s and the query size q. The evaluation time increases linearly with both the stream size s and the query size q. The effect is visible for both the real-life and the synthetic XML stream sets, with a higher increase for the latter due to the more complex data structure, cf. Figure 3.

Varying query characteristics. Figure 4 shows an increase of the evaluation time by a factor of less than 2 when p_{wild} and p_{desc} increase from 0 to 100%. It also suggests that the evaluation times for fsibl and child are comparable. Further experiments have shown that the evaluation of queries with predicates and set operations is slightly more expensive than the evaluation of simple path queries.

The **memory usage** has been almost constant over the full range of the experiments. Only an increase in query size q proved to have a noticeable effect: Increasing the query size from 1 to 1000 has led to an increase in memory usage of Java from 2 to 8 MB. This includes the memory needed for the transducer network and for its processing.

4. RELATED WORK AND CONCLUSION

There is a significant body of work in filtering and querying XML streams. Due to space reasons, only the general research directions are given here.

(1) In the context of publish-subscribe or event notification systems, the XML stream needs to be filtered by a large number of queries. In contrast to our work, the stream is assumed to be partitioned into comparatively small documents (in the range of hundreds to thousands of elements per document), and it is deemed sufficient to determine whether some queries match a document, rather than answering the queries. The filtering engines proposed recently, e.g., XFilter [1], construct an NFA for a bulk of queries, which is then transformed into a DFA. XTrie [2] introduces a novel query index structure for efficient query matching.

(2) Another direction is to provide an efficient single-pass XPath evaluator for single queries. XSM [5] proposes a network of finite-state transducers with buffers. Apart of the support for joins and element creation enabled by random-access buffers, the query language is severely restricted. Furthermore, it processes only streams with non-recursive structure definition. XSQ [11] introduces a hierarchical pushdown transducer, that provides good performance for restricted XPath (only child and desc axes, unnested predicates with at most one such axis), but only limited extensibility. Experiments with these systems show reasonable average-case performance, though the required space can grow exponentially in the query size.

The approach presented in the present paper is based on a network of simple, independent pushdown transducers that can be connected in a flexible manner. It allows not only for easy query language extensions, since adding new query constructs implemented by new transducers does not affect the existing ones, but also for extensive query optimization, e.g., by sharing transducers. Recent work of the authors [3] shows that this approach easily scales to large number of queries (e.g., tens of thousands of queries with average size 10) to be evaluated in a single-pass over the stream, as needed by publish-subscribe systems. This advantage is combined with a space and time complexity near the theoretical optimum for in-memory XPath evaluation [4], and a larger XPath fragment than in previous work, including all axes, structural (nested) predicates, and set operations.

References

[1] Mehmet Altinel and Michael J. Franklin, *Efficient filtering of XML documents for selective dissemination of information*, Proc. of VLDB, 2000.

[2] Chee-Yong Chan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi, *Efficient filtering of XML documents with XPath expressions*, VLDB Journal (2002).

[3] Tim Furche, *Optimizing multiple queries against XML streams*, Diploma thesis, Univ. of Munich, 2003.

[4] Georg Gottlob, Christoph Koch, and Reinhard Pichler, *XPath processing in a nutshell*, ACM SIGMOD Record **32** (2003).

[5] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou, *A transducer-based XML query processor*, Proc. of VLDB, 2002.

[6] Sam Madden and Michael J. Franklin, *Fjording the stream: An architecture for queries over streaming sensor data*, Proc. of ICDE, 2002.

[7] José M. Martínez, *MPEG-7 overview*, Tech. Report N4980, ISO/IEC JTC1/SC29/WG11, 2002.

[8] Gerome Miklau, *XMLData repository*, Univ. of Washington, 2003.

[9] Dan Olteanu, Tim Furche, and François Bry, *Evaluating complex queries against XML streams with polynomial combined complexity*, Tech. Report PMS-FB-2003-15, Univ. of Munich, 2003, http://www.pms.informatik.uni-muenchen.de/publikationen.

[10] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry, *XPath: Looking forward*, Proc. of EDBT Workshop XMLDM, 2002, LNCS 2490.

[11] Feng Peng and Sudarshan S. Chawathe, *XPath queries on streaming data*, Proc. of ACM SIGMOD, 2003.

[12] Dominik Schwald, *Approximate streamed evaluation of XPath under memory constraints*, Project thesis, Univ. of Munich, 2003.