Evaluating XPath Queries on XML Data Streams

Stefan Böttcher and Rita Steinmetz

University of Paderborn (Germany)
Computer Science
Fürstenallee 11 D-33102 Paderborn
stb@uni-paderborn.de, rst@uni-paderborn.de

Abstract. Whenever queries have to be evaluated on XML data streams - or when the memory that is available to evaluate the XML data is relatively small compared to the document - DOM based approaches that have to load and store large parts of the document in main memory will fail. In comparison, we present an approach to evaluate XPath queries on SAX streams that supports all axes of core XPath, including the sibling axes. Starting from the XPath query, our approach generates a stack of automata that uses the SAX stream as input and generates the result of the query as an output SAX stream. An evaluation of our implementation shows that in general our approach needs less main memory, but at the same time is faster than both, Saxon and YFilter.

1 Introduction

1.1 Motivation and Paper Organization

XML is becoming the de facto standard for information exchange and, as the amount of XML data is steadily growing, a key challenge is to process XML documents fast within the available main memory.

Our contribution focuses on scenarios, in which a system has to evaluate queries fast on documents that are multiple times larger than the main memory available to the system. One typical scenario is an XML news stream provided by a news agency using one of the typical XML formats NewsML [16] or NITF [17] to broadcast their news, and users who want to receive only parts of the news based on queries that represent their interests. Another typical scenario is that devices with a small amount of main memory (as e.g., mobile phones) shall work on large XML documents.

Whenever a scenario requires that the main memory available to evaluate queries on XML data is relatively small compared to the XML data size, approaches that are based on DOM will fail. These approaches have to load the complete XML document as a DOM tree into main memory, and as they need at least 4 pointers for each XML element (name, parent, first child, and next sibling) they yield a memory consumption that covers multiple times the size of the XML data.

Therefore, we propose a SAX based approach to the evaluation of XPath queries. Each input query is translated into an automaton that consists of only four different types of transitions, the treatment of which is described in Section 2. The small size of

R. Cooper and J. Kennedy (Eds.): BNCOD 2007, LNCS 4587, pp. 101–113, 2007.

the generated automata allows for a fast evaluation of the input XML data stream within a small amount of memory.

This paper is organized as follows: The remainder of the first section outlines the query language, summarizes the underlying assumptions, and outlines the problem definition. Section 2 summarizes the fundamental concepts used to describe our approach to evaluate XPath queries. The third section outlines some of the experiments that show the space efficiency and time efficiency of our prototype. Section 4 gives an overview on related work and is followed by the Summary and Conclusions.

1.2 Query Language

The subset of XPath expressions supported by our approach conforms to the set of *core XPath* as defined in [11]. This set is defined by the following EBNF grammar:

```
cxp         ::= `/' locationpath
locationpath      ::= locationstep ('/' locationstep)*
locationstep      ::= x `::' t | x `::' t `[' pred `]'
pred        ::= pred `and' pred | pred `or' pred
                | `not' `(' pred `)' | locationpath
                | locationpath ‘=’ const |`(' pred `)'
```

“cxp” is the start production, “x” represents an axis (attribute, self, child, parent, descendant-or-self, descendant, ancestor-or-self, ancestor, following, preceding, following-sibling, preceding-sibling), “const” represents a constant, and “t” represents a “node test" (either an XML node name test or “*”, meaning “any node name”).

Note that our system supports the sibling axes, whereas other approaches like XMLTK[1], χαοζ[4], AFilter[6], YFilter[9], XScan[14], SPEX[18], and XSQ[20] are limited to the parent-child and ancestor-descendant axes.

1.3 General Assumptions and Problem Definition

As our system is designed to efficiently evaluate XPath queries on a possibly infinite XML data stream, one requirement that our system has to meet is that each SAX event can be read only once, i.e., the stream has to be parsed in a single pass in document order. As we cannot jump backwards within the data stream, we have to rewrite user queries that use backward axes (i.e., ancestor-or-self, ancestor, preceding-sibling, and preceding) into equivalent queries containing only forward axes as described in [19]. The rewriting might lead to equivalent rewritten queries that are exponentially longer than the original queries, but as usually queries are rather short compared to the XML data, the growth of query length will usually not extend the runtime too badly.

Problem description: After rewriting queries, the remaining problem examined in this paper is the following. The input consists of a core XPath query containing only the forward axes and of an XML data stream in form of a SAX input event stream. The desired output is a SAX event stream of query results in document order. The main requirements of our system are to use as little main memory as possible in order to reach data throughput rates comparable to those of data streams.

2 Our Solution

In this section, we first explain how to transform the SAX input stream into a binary SAX event stream, containing firstchild::*, nextsibling::*, and parent::* events, and supporting self::a node tests. We then discuss how XPath queries are normalized, such that they contain only firstchild::*, nextsibling:*, and self::a location steps plus filters, and how normalized queries are transformed into XPath automata. Afterwards, we show how to evaluate the binary SAX event stream on an evaluation stack of an XPath automaton, which represents core XPath queries without any predicate filters. Finally, we extend the approach to queries with predicate filters.

2.1 Binary SAX Event Streams

We transform the SAX event stream of the input XML document into a stream of binary SAX events firstchild::*, nextsibling::*, parent::*, and self::a. Here, 'a' can be an element name, @ followed by an attribute name, or = followed by a constant. Transforming the SAX stream is done in two phases.

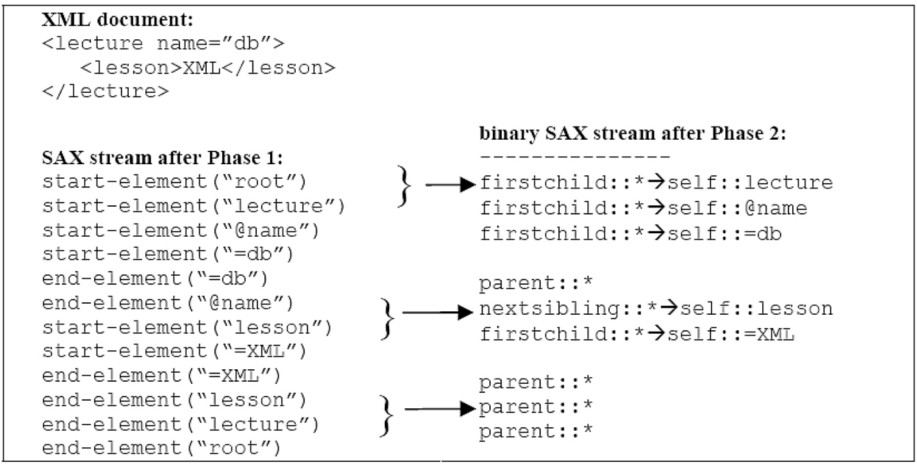

Fig. 1. Example XML document with the resulting SAX and binary SAX streams

Phase 1: The SAX event character(T) generated for a text value T found in the XML document is transformed into a binary SAX event sequence start-element(=T), end-element(=T). Similar, each attribute/value pair A=AV found in the XML document is transformed into a binary SAX event sequence

start-element(@A), start-element(=AV), end-element(=AV), end-element(@A) .

As the symbols '@' and '=' have to be chosen to uniquely identify attributes and text nodes respectively, they are not allowed as an initial character for element-names.

Finally, we replace the SAX event start-document with an event start-element("root"), and we replace the SAX event end-document with an event end-element("root"). At the end of Phase 1, the transformed SAX event stream contains only two kinds of events: start-element(…) and end-element(…).

Phase 2: For the replacement of all the start-element and end-element events with first-child::*, next-sibling::* or parent::* events, we regard the four different kinds of consecutive pairs of start-element and end-element events:

1. A start-element(x) followed by a second start-element(a) corresponds to the firstchild axis, i.e., 'a' is the first child of 'x'. Therefore, the event sequence firstchild::*→self::a is created.
2. An end-element(x) followed by a start-element(a) corresponds to the nextsibling axis, i.e., 'a' is the next sibling of 'x'. Therefore, the event sequence nextsibling::*→self::a is created.
3. Furthermore, an end-element(x) followed by a second end-element(y) corresponds to the parent axis. Therefore, the event parent::* is created.
4. When a start-element(x) is followed by an end-element(x), no binary SAX event is created.

Altogether, Phase 1 and Phase 2 together transform a SAX stream into a binary SAX stream of firstchild::*, nextsibling::*, parent::*, and self::a events. Figure 1 presents an example of an XML document and the generated binary SAX event stream.

The binary SAX events are used as input 'symbols' for a stack of XPath automata that is constructed for an XPath query as described in the following sub-sections.

2.2 Decomposition and Normalization of XPath Query Expressions

We decompose each XPath query into a set of filter-free *path queries*, and, corresponding to the transformation of the SAX input stream, rewrite each path query into an equivalent XPath expression, called *normalized XPath expression*, that contains only the location steps firstchild::*, nextsibling::*, and self::a. Here, 'a' can be an element name, @ followed by an attribute name, or = followed by a constant as in binary SAX events, but 'a' can also be the wildcard '*' for an arbitrary node name.

Step 1 (Decomposition): We recursively decompose each XPath query Q into a set of filter-free sub-queries, called *query paths*, by decomposing Q into the main path M and predicate paths P1,…, Pn. A predicate path Pi of the form path = const is rewritten to path/text::const.

For example, an XPath query
`Q=/descendant::a[child::b=xyz]/child::c[child::d/child::e]/f` is decomposed into 3 query paths: the *main path*
`M=/descendant::a/child::c/child::f`
and the *predicate paths* `P1=child::b/text::xyz` and `P2=child::d/child::e`.

Step 2 (Normalization): After decomposing `Q`, each of its query paths `M, P1, …, Pn` is normalized separately as follows. We replace the axes `following, descendant-or-self`, `attribute`, and `text` according to the following rewrite rules:

```
(1) following::a    ➔ ancestor-or-self::*/following-sibling::*
                                          /descendant-or-self::a
(2) descendant-or-self::a  ➔   descendant::a | self::a
(3) attribute::a           ➔   child::@a
(4) text::v                ➔   child::=v
```

Note, that the disjunction (|) in rule (2) does not lead to an exponential growth of the query size, but only to one additional edge in the XPath automaton (c.f. Figure 2).

As the rewrite rule (1) which replaces the following axis leads to an ancestor-or-self axis, we eliminate the backward axis ancestor-or-self according to the rewrite rules (13)-(22) provided in [19]. As the result of Step 2, we get an equivalent XPath query that contains only the axes self, child, descendant, and following-sibling.

2.3 Transforming a Filter-Free XPath Query into an XPath Automaton

In order to evaluate a query path, we first build an XPath automaton and then start the XPath evaluation stack using this automaton and the binary SAX stream as input.

Definition 1 (XPath automaton): An XPath automaton of a query path is a NFA

$$XP = (Q, \Sigma, q0, \delta, f), \text{ where}$$

- Q is the finite set of states
- Σ={firstchild::*, nextsibling::*} $\cup$ {self::a | a is an element name, @ followed by an attribute name, = followed by a constant or '*' } is the set of input symbols
- $q0 \in Q$ is the start state
- $\delta : Q \times \Sigma \times Q$ is a relation of *transitions* (q1,e,q2) where q2 is a successor state of q1 if the event e is sent to the NFA,
- $f \in Q$ is the final state

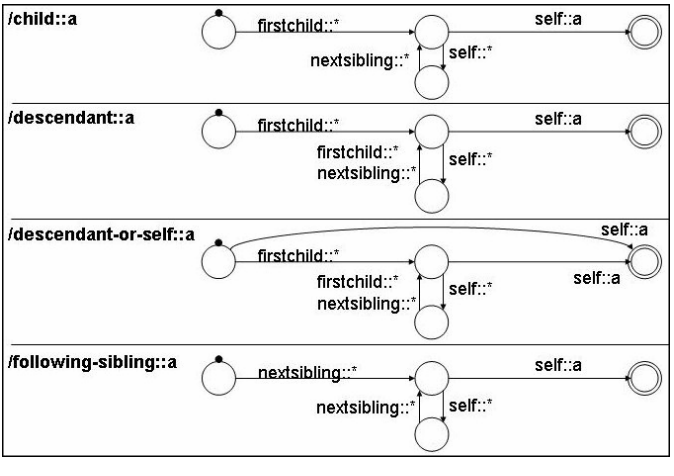

Fig. 2. Atomic XPath Automata

In order to build the XPath automaton for a given query path, we normalize each location step as described in Step 2 of Section 2.2. After normalization, we compute the so-called *atomic XPath automaton* for each location step. The atomic XPath automata for the location steps '/child::a', '/descendant::a', '/descendant-or-self::a', and '/following-sibling::a' are shown in Figure 2.

An atomic XPath automaton of the child axis, the descendant axis or the following-sibling axis location step (c.f. Figure 2) is an automaton that is equivalent to the regular expression that forms the right-hand-side of the following rewrite rules (which were inspired by [11]) for the corresponding location step.

```
(5) child::a➔  firstchild::*/
            (self::*/nextsibling::*)^i/ self::a            0≤i<∞
(6) descendant::a   ➔  firstchild::*/ (self::*/(firstchild::* |
                              nextsibling::*)) ^i/self::a   0≤i<∞
(7) following-sibling::a➔  nextsibling::*/
                      (self::*/nextsibling::*)^i/self::a   0≤i<∞
```

The right hand sides of the rules (5)-(7) correspond to regular expressions over the alphabet $\sum$ of input symbols given in Definition 1, and the exponent 'i' corresponds to the kleene star operator in regular expressions. We have used the exponent i to avoid disambiguities between the kleene star operator for regular expressions and the (wildcard) *-operator in XPath expressions.

Note that the location step 'self::*' is inserted into the right hand sides of the rules (5)-(7), such that both the 'firstchild::*' and the 'nextsibling::*' location steps are followed by a self axis location step, which corresponds to the sequence of events of a binary SAX event stream as described in Section 2.1.

The complete XPath automaton of a query path is built by concatenating the atomic XPath automata of all the query path's locations steps in the order given by the location steps. To concatenate the atomic XPath automata ALS1 and ALS2 of two location steps LS1 and LS2 into a new XPath automaton XLS means to combine the final state of ALS1 with the start state of ALS2 to a single state. The start state of the XLS is the start state of ALS1 and the final state of XLS is the final state of ALS2.

Whenever the final state of the XPath automaton representing the main path is reached, we have reached a part of the answer, and the current "sub-tree" of the binary SAX stream is written to the SAX output stream.

2.4 Evaluating Filter-Free XPath Queries Using XPath Automata

Definition 2 (XPath evaluation stack): An *XPath evaluation stack of an XPath automaton XP* is a triple

$$XPE = (XP, \sum, \Delta) \text{ with}$$

- XP is used as the initial stack symbol
- $\sum$ = {firstchild::*, nextsibling::*, parent::*} $\cup$ {self::a | a is an element name, @ followed by an attribute name, = followed by a constant or '*'} is the set of input symbols
- $\Delta(\sum)$ is an evaluation function that performs for a given input symbol a

sequence of operations
Δ(firstchild::*) = {push(top()); top().event(firstchild::*);}
Δ(nextsibling::*) = {top().event(nextsibling::*);}
Δ(parent::*) = {pop();}
Δ(self::a) = { closure(top().event(self::a)); }

The operation 'XP Stack. top()' returns the XPath automaton on top of the stack, and the operation 'void XP.event(InputSymbol)' fires the event InputSymbol on the XPath automaton XP. The operation 'void Stack.push(XP)' puts the XPath automaton XP on top of the stack, such that Δ(firstchild::*) pushes a copy of the XP automaton that is the top stack element on top of the stack and passes the event firstchild::* to this copy. The operation 'void Stack.pop()' deletes the XPath automaton on top of the stack. Finally, the closure-operator in Δ(self::a) sends an event self::a to the automaton stored at top of stack as often as the state of this automaton changes.

Evaluation of filter-free XPath queries: Each filter-free XPath query X is evaluated on a stream of binary SAX events S as follows. We compute the XPath automaton XP of X and start the XPath evaluation stack with XP as initial stack symbol and with S as input. Each binary SAX event is passed as input symbol to the stack, and the Δ(∑) function is performed for this input symbol which eventually causes stack operations and events on an automaton stored in the stack.

Whenever a final state of an XPath automaton that is stored on top of stack is reached, the XML sub-tree with the root element that corresponds to the SAX event last parsed is written to the SAX output stream.

Optimized implementation: As all XPath automata stored on the stack share the same structure, i.e., Q, ∑, q0, δ, and f are identical for all automata of the stack, in our implementation, we do not store and copy automata. Instead, there exists one global XPath automaton, and the stack stores only the set of active states on each level.

2.5 Evaluation of Automata for XPath Expressions with Predicate Filters

Whenever a location step LS contains a predicate filter, after query decomposition, a filter automaton F is created for the predicate path P corresponding to the filter, and F is attached to the final state fls of the atomic automaton of LS. A *filter automaton F* is an XPath automaton, but F's final state does not cause any output.

Whenever the state fls is reached by firing a transition, a so-called *reservation* is created and attached to fls and the start state of the attached filter automaton becomes active too, i.e., all binary SAX events are regarded as input for both the main automaton and the filter automaton. Each *reservation* is a Boolean variable, which will evaluate to either true as soon as the filter automaton has reached its final state or to false as soon as the automaton in which this filter automaton became active is popped from the stack.

More precisely, reservations are computed as follows. Let R, R1, R2 be sets of reservations, and let res: Q x ℘(R) be a mapping of XPath automaton states to sets of reservations. Each XPath automaton XP used in the XPath evaluation stack is initialized without any reservations, i.e., ∀ q∈ XP.Q: res(q, {}). Whenever a state q is

reached in XP, and a filter automaton F is attached to q, the mapping is changed from res(q,R) to res(q,R$\cup${r}), where r is a new reservation generated for F. Furthermore, when a transition of the form δ(q1,inputSymbol,q2) is fired, all reservations R1 for a state q1 become also reservations of the state q2 of XP. To summarize, the set of reservations R2 of q2 is R2=R1$\cup${r1,…,rf}, where r1,…,rf are the newly created reservations for the filter automata attached to q2.

If the final state f of an XPath automaton of the main path of the given XPath query is reached, and there exists a reservation r that is attached to f that is not yet evaluated, the output of the current sub-tree is queued and delayed until the reservation r is evaluated; the current sub-tree becomes an output *candidate*. Finally, when r is evaluated to true, the sub-tree is written to the output and deleted from the queue. If on the other hand r is evaluated to false, the sub-tree is deleted from the queue without writing it to the output.

A reservation r evaluates to true, if the corresponding filter automaton F reaches a final state. In this case, r is set to true, possibly queued sub-trees can be written to the output. If the automaton in which F became active is popped from the stack and no final state of F has been reached in the meantime, the reservation r for F evaluates to false, and possibly queued sub-trees that carry the reservation r are deleted without being written to the output and all active states s with res(s,R), r$\in$R, become inactive.

As a predicate filter can not only contain a single comparison path=value, but can be a composition of comparisons involving nested negations, disjunctions or conjunctions of comparisons, reservations can be logical compositions of sub-reservations, too. For example, a predicate filter [(comp1 or comp2) and not comp3], where comp1, comp2 and comp3 are comparisons or path expressions, results in a composed reservation r = ((r1 or r2) and not r3) and a filter automaton being created for each sub-reservation r1, r2, and r3.

Simple and composed reservations are administrated in a *lemma table*. Whenever a reservation is evaluated, the result is reported to the lemma table. The lemma table is used for checking whether a composed reservation can be evaluated completely, i.e., whether the lemma table knows enough results of sub-reservations to decide, whether the value of the composed reservation is true or false. The lemma table reports the value of the evaluated reservation back to the XPath automaton XP waiting for the reservation, such that XP can continue processing, and finally the main automaton can check the output queue, and output candidates might be written to the output.

3 Evaluation of Our Prototype Implementation

We have implemented a prototype of our solution (XPA) in Java 1.5 and have evaluated and compared it with two other systems on a Pentium 4 with 2.4 GHz Windows XP system with 1 GB of RAM running Java 1.5. On the one hand, we have compared XPA with the static XPath evaluator Saxon[21] that is DOM based, and therefore is not capable to evaluate data streams. On the other hand, we have compared XPA with YFilter[9], a system for information dissemination that is designed to evaluate a set of queries on large XML data streams.

Our test data set was generated by the XML generator of the XML Benchmark XMark[22]. The sizes of the documents of our data set can be seen in Table 1. A document Dn was created by the XMark generator providing the factor n/1000, i.e., D32 was generated by the XMark generator with the factor 0.032. This leads to a dataset with documents starting from the size of 116 kB to the size of more than 650 MB.

Table 1. Document sizes of the test collection (generated by XMark)

Document name	D1	D2	D4	D8	D16	D32	D64	D128	D256	D512	D1024	D2048	D4096	D6000
Document size (kB)	116	211	458	901	1,881	3,728	7,259	14,949	29,693	59,114	118,767	238,164	477,018	697,657

On our dataset, we have evaluated queries that were inspired by the queries Q1,…,Q5 of the XPath benchmark XPathMark[10] (we have omitted all backward axes in advance). The test queries can be seen in Table 2.

Table 2. XPathMark queries used for the evaluation of the XPath evaluation system XPA

Name	Query
Q1	/child::site/child::regions/child::*/child::item
Q2	/child::site/child::closed_auctions/child::closed_auction/child::annotation/ child::description/child::parlist/child::listitem/child::text/child::keyword
Q3	/descendant::keyword
Q4	/descendant-or-self::listitem/descendant-or-self::keyword
Q5	/child::site/child::regions/child::*[self::namerica]/child::item

Our tests have shown that our system outperforms the other two systems. Especially for large documents, our system is more than 2 times faster than Saxon and 20 times faster than YFilter. Table 3 shows the concrete figures for the query Q5. A visualization of the figures for the query Q5 can be seen in Figure 3(a), whereas Figure 3(b) and 3(c) show the evaluation times for all queries for document D1 or D1024 respectively.

Our tests have as well shown that our system consumes far less main memory than Saxon and than YFilter. Saxon consumes 4 times the document size on average, which is typical for DOM based systems, YFilter needs only 2 times the document size. In comparison, XPA consumes from 20% of the document size on average for simples XPath queries without predicate filters (Q1-Q4) up to 50 % of the document size on average for paths with predicate filters (Q5). In our experiments, an OutOfMemory-Exception for YFilter occured from D2048 on and for Saxon from D4096 on with 1 GB of heap space assigned to Java.

On average, we have measured a data throughput rate of more than 40MBit/s for our system. In comparison, ADSL2+, the fastest ADSL standard currently available, reaches a data download throughput rate of at most 24 MBit/s.

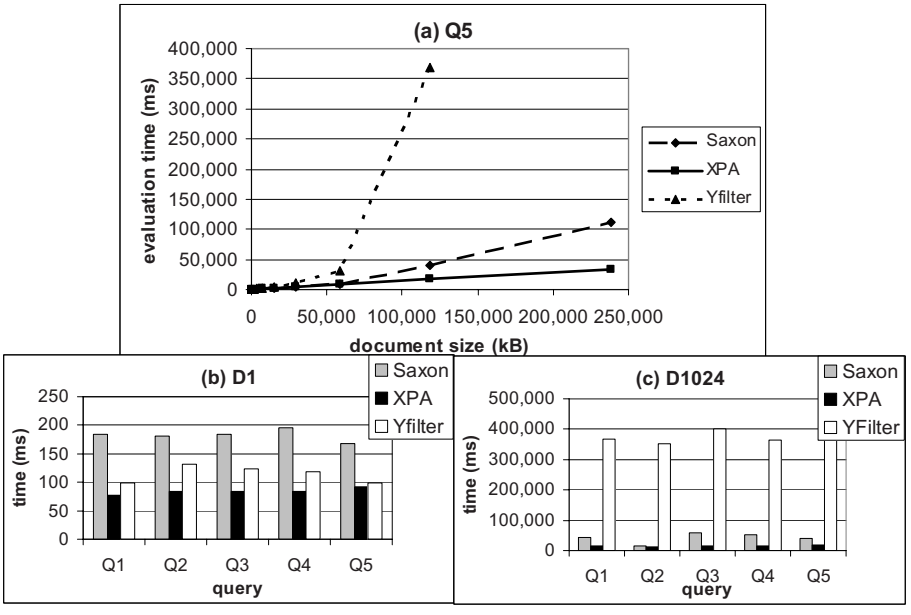

Fig. 3. (a) Evaluation time for different document sizes (query: Q5). (b) Evaluation time for document D1 for all queries. (c) Evaluation time for D1024 for all queries.

Table 3. Evaluation time for different document sizes (query Q5) (□ = OutOfMemory)

	D1	D2	D4	D8	D16	D32	D64	D128	D256	D512	D1024	D2048	D4096	D6000
XPA	92	123	136	219	383	688	1,204	2,288	4,194	8,623	16,983	32,945	109,471	186,282
Saxon	168	171	257	366	651	898	1,511	2,482	4,500	9,515	40,403	111,862	∅	∅
YFilter	99	159	256	422	778	1,387	2,640	5,102	10,115	30,914	367,790	∅	∅	∅

4 Relation to Other Works

There exist several different approaches to the evaluation of XPath queries on XML data streams. They can be divided into categories by the subset of XPath they support. Nearly all of them are based on automatons (X-scan[14], XMLTK[1], YFilter[9], [12], [13], AFilter[6], XSQ[20], SPEX[18]) or parse trees ([3], [4], [7], [8]). All of them support the axes child and descendant-or-self and most of them support predicate filters and wildcards, but none of them support the sibling-axes as our solution does.

X-scan[14], XMLTK[1], and YFilter[9] support XPath queries containing the child and descendant-or-self axes and wildcards using finite state automata. [12] (for the main path) and [13] (for the predicates) propose to construct deterministic finite automata (DFA) in a lazy way, i.e., the DFA is not generated completely at the beginning, but additional states are added only when needed.

AFilter[6] is an adaptable XPath query evaluation approach that needs a base memory requirement that is linear in query and data size. If more memory is provided to AFilter, AFilter uses the remaining main memory for a caching approach to evaluate queries faster than with only the base memory. AFilter is mainly based on a lazy DFA and it supports wildcard, but does not support predicate filters. Similar to YFilter[9], AFilter is designed to evaluate a large set of queries.

XSQ[20] and SPEX[18] use a hierarchical arrangement or network of transducers, i.e., automata extended by actions attached to the states, extended by a buffer to evaluate XPath queries. The XPath queries supported by XSQ contain predicates with the restriction that each query node can contain at most one predicate and each predicate can contain path-to-value comparisons with paths of size 1 containing only the axes child, text or attribute. The main idea is that a nondeterministic push-down transducer (PDT) is generated for each location step in an XPath query, and these PDTs are combined into a hierarchical pushdown transducer in the form of a binary tree.

The approach presented in [15] discusses how to handle the child and descendant-or-self axes, predicates (including functions and arithmetics) and wildcards in XQuery using TurboXPath. The input query is translated into a set of parse trees. Whenever a matching of a parse tree is found within the data stream, the relevant data is stored in form of a tuple that is afterwards evaluated to check whether predicate- and join conditions are fulfilled. The output is constructed out of those tuples the conditions of which have been evaluated to true.

χαοζ[4] and [3] build a parse tree as well (plus a parse-dag in [4], as they support the parent and the ancestor axis in addition). These parse tree is used to 'predict' the next matching nodes and the level in which they have to occur. For example, consider the query //a/b and a matching of 'a' in level 3. Then the next interesting matching would be a node 'b' in level 4.

The approach discussed in [7] is mainly based on parse trees, but it collapses the parse tree into a prefix trie as follows. Common prefix sequences of child-axis location steps of different queries are combined into a leaner single path of the prefix trie.

The approach presented in [8] uses a structure which resembles a parse tree with stacks attached to each node. These stacks are used to store XML nodes that are solutions to the parse tree nodes subquery (or to store XML nodes that are candidates for a solution in case of predicate filters).

In comparison to all these approaches, we additionally support the 'sibling'-axes following and following-sibling. Furthermore, beyond [15] and [20], our approach is capable to parse streams of recursive XML, i.e., data in which the same element names do occur repeatedly along a root-to-leaf path.

5 Summary and Conclusions

Query processing on massive XML data streams and query processing of XML data on small mobile devices require a query processor to meet two conditions at the same time: the query processor shall consume a small amount of main memory and shall reach data throughput rates that are not smaller than the arrival rate of the XML data using today's broadband communication technologies.

In this paper, we have presented an XPath query processor that reaches data throughput rates that are higher than the download rates of ADSL2+ while at the same time consuming only 20%-50% of the document size in main memory. Furthermore, in comparison to most of the other query processors, our query processor supports all the axes of core XPath including the sibling axes.

Our query processor decomposes and normalizes each XPath query, such that the resulting path queries contain only three different types of axes, and then converts them into lean XPath automata for which a stack of active states is stored. The input SAX event stream is converted into a binary SAX event stream that serves as input of the XPath automata.

As XPath is used as data access standard in XSLT and XQuery, we are optimistic that the technology proposed in this paper can be used within XSLT processors or XQuery processors too.

References

[1] Avila-Campillo, I., Green, T.J., Gupta, A., Onizuka, M., Raven, D., Suciu, D., XMLTK,: An XML Toolkit for Scalable XML Stream Processin. In: Proceedings of PLANX (October 2002)

[2] Bar-Yossef, Z., Fontoura, M., Josifovski, V.: Buffering in query evaluation over XML streams. PODS 2005, pp. 216–227 (2005)

[3] Bar-Yossef, Z., Fontoura, M., Josifovski, V.: On the Memory Requirements of XPath Evaluation over XML Streams. PODS 2004, pp. 177–188 (2004)

[4] Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M., Josifovski, V.: Streaming XPath Processing with Forward and Backward Axes. ICDE 2003, pp. 455–466 (2003)

[5] Bry, F., Coskun, F., Durmaz, S., Furche, T., Olteanu, D., Spannagel, M.: The XML Stream Query Processor SPEX. ICDE 2005, pp. 1120–1121 (2005)

[6] Candan, K.S., Hsiung, W.-P., Chen, S., Tatemura, J., Agrawal, D.: AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering. VLDB 2006, pp. 559–570 (2006)

[7] Chan, C.Y., Felber, P., Garofalakis, M.N., Rastogi, R.: Efficient Filtering of XML Documents with XPath Expressions. ICDE 2002, pp. 235–244 (2002)

[8] Chen, Y., Davidson, S.B., Zheng, Y.: An Efficient XPath Query Processor for XML Streams. In: Proceedings of 22nd International Conference on Data Engineering (ICDE) (to appear, 2006)

[9] Diao, Y., Rizvi, S., Franklin, M.J.: Towards an Internet-Scale XML Dissemination Service. In: Proceedings of VLDB 2004 (August 2004)

[10] Franceschet, M.: XPathMark: An XPath Benchmark for the XMark Generated Data. In: Bressan, S., Ceri, S., Hunt, E., Ives, Z.G., Bellahsène, Z., Rys, M., Unland, R. (eds.) XSym 2005. LNCS, vol. 3671, pp. 129–143. Springer, Heidelberg (2005)

[11] Gottlob, G., Koch, C., Pichler, R.: Efficient Algorithms for Processing XPath Queries. VLDB 2002 (2002)

[12] Green, T.J., Gupta, A., Miklau, G., Onizuka, M., Suciu, D.: Processing XML Streams with Deterministic Automata and Stream Indexes Published in ACM TODS, vol. 29(4), pp. 752–788 (December 2004)

[13] Gupta, A., Suciu, D.: Stream Processing of XPath Queries with Predicate. In: Proceeding of ACM SIGMOD Conference on Management of Data (2003)

[14] Ives, Z.G., Halevy, A.Y., Weld, D.S.: An XML query engine for network-bound data. VLDB J. 11(4), 380–402 (2002)
[15] Josifovski, V., Fontoura, M., Barta, A.: Querying XML streams. VLDB J. 14(2), 197–210 (2005)
[16] NewsML 1.2: News Markup Language (October 2003) http://www.newsml.org/
[17] NITF 3.3: News Industry Text Format, http://www.nitf.org/
[18] Olteanu, D., Kiesling, T., Bry, F.: An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. ICDE 2003, pp. 702–704 (2003)
[19] Olteanu, D., Meuss, H., Furche, T., Bry, F.: XPath: Looking Forward. EDBT Workshops 2002, pp. 109–127 (2002)
[20] Peng, F., Chawathe, S.S.: XPath Queries on Streaming Data. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. June 9-12 2003, San Diego, California (2003)
[21] SAXON - XSLT and XQUERY Prozessor Version 8.8.0.4. 2006 http://saxon.sourceforge.net/
[22] Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. VLDB 2002, pp. 974–985 (2002)