Mixing Bottom-Up and Top-Down XPath Query Evaluation

Markus Benter, Stefan Böttcher, and Rita Hartel

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

Abstract. Available XPath evaluators basically follow one of two strategies to evaluate an XPath query on hierarchical XML data: either they evaluate it top-down or they evaluate it bottom-up. In this paper, we present an approach that allows evaluating an XPath query in arbitrary directions, including a mixture of bottom-up and top-down direction. For each location step, it can be decided whether to evaluate it top-down or bottom-up, such that we can start e.g. with a location step of low selectivity and evaluate all child-axis steps top-down at the same time. As our experiments have shown, this approach allows for a very efficient XPath evaluation which is 15 times faster than the JDK1.6 XPath query evaluation (JAXP) and which is several times faster than MonetDB if the file size is ≤ 30 MB or the query to be evaluated contains at least one location step that has a low selectivity. Furthermore, our approach is applicable to most compressed XML formats too, which may prevent swapping when a large XML document does not fit into main memory but its compressed representation does.

Keywords: XML, top-down XPath evaluation, bottom-up XPath evaluation.

1 Introduction

1.1 Motivation

XML gains more and more popularity not only as a data exchange format, but also as a storage, archive or data management format and XPath is the main standard to express path queries on XML data.

Whenever XPath query evaluation is a bottleneck of an application, a fast XPath query evaluator is desired. If in addition, XML documents may become larger than the available main memory space, it may be a significant advantage when the fast XPath query evaluator can process XPath queries on compressed XML documents that can still fit into main memory. We present such a fast XPath query evaluator that relies on just a minimal set of XML navigation steps, such that it is applicable not only to plain XML data, but also to most queryable compressed XML data formats.

J. Eder, M. Bielikova, and A.M. Tjoa (Eds.): ADBIS 2011, LNCS 6909, pp. 27–41, 2011.
© Springer-Verlag Berlin Heidelberg 2011

1.2 Contributions

"Traditional" XPath evaluators typically evaluate the hierarchical XML data either top-down or bottom-up, as both techniques provide advantages for different classes of queries. In this paper we present an approach that allows XPath evaluation in any direction and that combines the following properties:

- The approach presented in this paper supports both, bottom-up and top-down XPath query evaluation on an XPath subset that extends *core XPath* as defined in [1] by comparisons of paths to constants within predicate filters.
- Even more, our approach allows a dynamic mixture of bottom-up and top-down query evaluation, such that for each location step, it can be decided whether to evaluate it top-down or bottom-up and at which time of the query evaluation process.
- Our approach is powerful and generic as it requires only minimal support from the underlying XML format. That is, our approach can be applied to any uncompressed or compressed XML representation that provides access to XML nodes via the node's name and provides navigation via the binary axes first-child, first-child^{-1}, next-sibling, and next-sibling^{-1}, and nevertheless, our approach supports all the other XPath axes of core XPath (e.g. ancestor, descendant, following and preceding) within XPath queries.
- We have evaluated query performance on two different XML representations – one uncompressed and one compressed – that are integrated into our approach. Besides a DOM-based XML representation, we have implemented a second XML main-memory representation, that is based on Succinct compression [2] and that – if combined with an index – not only allows for an XPath evaluation as fast as the DOM-based representation, but also needs only 20% of the main memory required by a DOM representation.
- Finally, we have implemented different 'evaluation strategies' that decide, which sub-queries of a given XPath query to evaluate in which direction, i.e. top-down or bottom-up, and at which time of the evaluation process. Furthermore, we have evaluated and compared these navigation strategies within a series of experiments to determine which is the most efficient navigation strategy to evaluate XPath queries. Our experiments have shown that for our test queries, the mixed approach is up to 7 times faster than bottom-up evaluation and up to 56 times faster than top-down evaluation.

1.3 Query Language

The subset of XPath expressions supported by our approach extends the set of *core XPath* as defined in [1], as our approach beyond [2] additionally allows comparisons of paths to constants within predicate filters. This XPath subset supported by our approach 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 symbol, "x" represents an axis (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 – aside from the evaluation in top-down or in bottom-up direction – using the sibling axes in XPath queries, whereas other approaches like XMLTK[3], χαοζ[4], AFilter [5], YFilter[6], XScan[7], SPEX[8], and XSQ[9] are limited to using the parent-child and the ancestor-descendant axes only.

1.4 Paper Organization

This paper is organized as follows: Section 2 summarizes the fundamental concepts used for describing our approach to evaluate XPath queries consisting of a single path and XPath queries with filters. Furthermore, this section describes the different evaluation strategies that could be used for evaluating an XPath query. The third section outlines some of the experiments that compare the different evaluation strategies of our prototype with each other and with other XPath evaluators. Section 4 gives an overview of related work and is followed by the Summary and Conclusions.

2 Our Solution

2.1 Overview of Our Solution

We follow the ideas of [1] and [10] to rewrite the given XPath queries, such that they no longer use all the core XPath axes, but only a small set of basic binary axes containing the axes first-child, first-child^{-1}, next-sibling, next-sibling^{-1}, and self. Table 1 shows how to rewrite each standard XPath axis into a regular expression using only the basic binary axes.

Table 1. Axis definition in terms of the basic binary axes

Axis	Binary expression
child	first-child, (next-sibling)*
parent	(next-sibling^{-1})*, first-child^{-1}
descendant	first-child, (first-child | next-sibling)*
ancestor	(first-child^{-1} | next-sibling^{-1})*, first-child^{-1}
following-sibling	next-sibling, (next-sibling)*
preceding-sibling	next-sibling^{-1}, (next-sibling^{-1})*
following	(((first-child^{-1} | next-sibling^{-1})*,first-child^{-1}) | self), next-sibling, (next-sibling)*, ((first-child, (first-child | next-sibling)*) | self)
preceding	(((first-child^{-1} | next-sibling^{-1})*,first-child^{-1}) | self), next-sibling^{-1}, (next-sibling^{-1})*, ((first-child, (first-child | next-sibling)*) | self)

Based on the binary XPath expressions given in Table1, we provide an atomic automaton using the binary axes for each XPath axis. For example, Figure 1 (a) shows an automaton generated for a location step child::a, and Figure 1 (b) shows an automaton generated for a location step parent::a. fc represents the first-child axis, fcR the axis first-child^{-1}, ns the next-sibling axis, nsR the axis next-sibling^{-1}, and self the self axis.

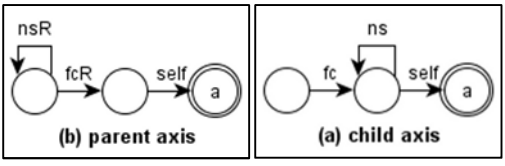


Fig. 1. Atomic automata for the location steps (a) child::a and (b) parent::a

Similar to the approach provided in[10], we translate each location step LSi of a given XPath query into a single atomic automaton BAi. The advantage of reducing all the XML axes listed in Table 1 to the basic axes (first-child, first-child^{-1}, next-sibling, next-sibling^{-1}, and self) is that we require the XML representation only to support the navigation along the basic axes together with an efficient access to all nodes that have a given node name. This requirement is met e.g. for uncompressed XML by the DOM representation or e.g. for compressed XML by the succinct representation[2]. Therefore, the presented approach can be applied to uncompressed XML as well as to compressed XML, if the XML format supports at least navigation along the basic axes and access to node names, although our approach applied to the XML format supports the much larger superset of core XPath described in Section 1.3.

The XPath query is represented as a special kind of non-deterministic finite automaton that we call a 'token automaton'. A token automaton not only contains states and transitions, but also allows for using each state in any number of tokens each of which represents an answer to a sub-query within the XML document. According to the events produced by the input XML document representation, the token-automaton fires transitions and transfers tokens, i.e. generates new tokens, along the binary axes first-child, first-child^{-1}, next-sibling, next-sibling^{-1}, and self.

The atomic token automata build a construction kit from which the final automaton representing an XPath query is built. In contrast to traditional automata, not the input – i.e., the XML document representation – controls, which transitions can be fired next, but there exist an external controlling instance – called *DecisionModule* – that decides, which transition will be fired next. In other words, the DecisionModule decides for each location step of the query whether it is evaluated top-down or bottom-up, and at which time of the query evaluation this location step is evaluated.

Each transition of the automaton can either be fired top-down, i.e., it consumes the binary axis that is denoted by the transition label and the tokens are transferred in the direction given by the transition, or it can be fired bottom-up, i.e., it consumes the inverse of the binary axis denoted by the transition label and the tokens are transferred opposite to the given direction. We assume, that the used XML compression provides – similar as it is provided by DOM – access to a list of nodes that fulfill a given node name test and supports navigation via the binary axes first-child, first-child^{-1}, next-sibling, next-sibling^{-1}, and self.

2.2 XPath Automata

Each atomic automaton contains one state that is called a stable state and that carries a node name test as label and that accepts only the tokens referring to those XML nodes which fulfill the given node name test. Stable states are marked by a double circle.

The notation of the transitions of the automaton only shows the top-down evaluation; the bottom-up evaluation can be taken by reversing the transition direction and by replacing each transition label by its reverse. The other atomic automata are built in a similar way to the child::a automaton shown in Fig. 1 according to the regular expressions provided in Table 1. If a location step LSi is followed by a location step LSj in a query Q, we concatenate the atomic automaton BAi corresponding to LSi and the atomic automaton BAj corresponding to LSj to the token automaton XPQ of query Q by drawing a self transition from the final state of BAi to the start state of BAj.

For example, Fig. 2(a) shows the automaton for the query Q = //a/b. All states have as label an ID of the form s0,…,s6, and the stable states have as an additional label the node name test that has to be fulfilled by an XML node in order to be accepted by this stable state. The root (state s0) is connected via a self-axis to the automaton for //a (states s1-s3) which is connected by another self-axis to the automaton for /b (states s4-s6).

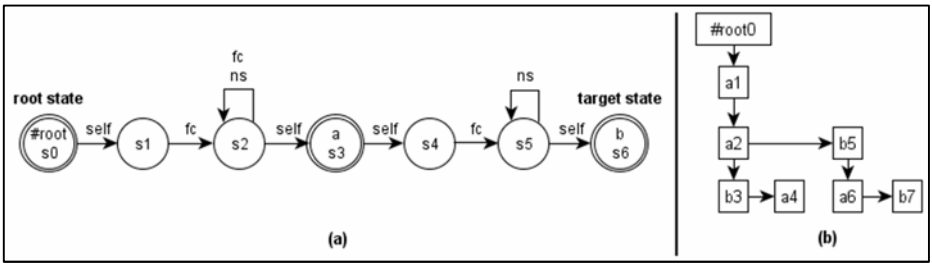


Fig. 2. (a) XPath automaton for query //a/b , and (b) a small example document where each node is represented by a node ID and the node's label

2.3 Evaluation of Filter-Less Paths

Overview: In order to evaluate an XPath query Q, first, the automaton A for Q is built as described in the previous section. Each pair (sx,sy) of stable states in A for which a path from sx to sy exists in A represents a relative XPath expression R that is a sub-sequence of location steps of Q.

Second, tokens each of which represents an answer to such a relative XPath expression R are created, transferred, joined, and deleted until all tokens that represent an answer to Q are computed. Let sx, sy be stable states in A, let R be the sub-query of Q that corresponds to the sub-automaton including all paths and states from sx to sy of the automaton A for Q, and let nv, nw be nodes in the given XML document. Then the token T=(nv/sx, nw/sy) represents an answer nw to the sub-query nv/R .

DecisionModule: A DecisionModule controls the order and the direction (bottom-up or top-down), in which the sub-queries are evaluated, i.e., it decides which of the stable states are taken as start states and for which state occurring in a token, partial sub-query evaluation is continued, i.e., which tokens are transferred next in which direction, and when tokens are joined.

Token Creation in Start States: Start states can be defined at any time during the execution. At any time, each state, none of the incoming or outgoing transitions of which had been fired, can be chosen as an additional start state. Whenever the DecisionModule declares a stable state s to be a start state, for each node n in the XML document that fulfills the node name test of s, a token (n/s, n/s) is created.

If for example the state with ID s6 of Fig. 2(a) is declared as a start state, i.e., we look for XML elements that are answers to the sub-query //self::b, tokens (3/s6, 3/s6), (5/s6, 5/s6), and (7/s6, 7/s6) are created for the nodes b3, b5, and b7 in Fig. 2(b), where b is the node label followed by the node ID (3,5,or 7).

Token Deletion: For each state s in A, except for the root state and the target state, when all transitions from s and all transitions to s have been fired and all join tokens for s have been computed as described below, all tokens containing s as the start state or as the final state are automatically deleted. Automatic token deletion can be partially switched off in order to implement a navigation cache as described in Section 2.4.

Partial Sub-query Evaluation: A sub-query R is top-down partially evaluated by firing all the transitions on a path from sx to sy. This operates on all tokens T=(nu/sz, nv/sx) that contain sx as their final state, i.e. represent an answer to a sub-query represented by paths in A ending in state sx, and it eventually generates new tokens that contain sy as their final state, i.e., for each answer nw to nv/R, this generates a new token T'=(nu/sz, nw/sy).

Similarly, a sub-query is bottom-up partially evaluated by firing all the inverted transitions of the transitions on a path from sx to sy in reversed order. This operates on all tokens T=(nw/sy, nt/sz) that contain sy as their start state, and it eventually generates new tokens that contain sx as their start state, i.e., for each answer nv to nw/R^{-1}, this generates a new token T'=(nv/sx, nt/sz).

Let si and sj be stable or non-stable states. To fire a transition with label fc (or ns or fcR or nsR respectively) that starts in state si and ends in state sj for a token T=(nu/sz, nv/si) with final state si in top-down direction means the following: to check, whether there exists a node with ID nw in the XML tree such that the node with ID nw is the first-child (or the next-sibling or fcR or previous-sibling respectively) of the node with ID nv. If such a node exists, a token T'=(nu/sz, nw/sj) is generated, otherwise no token is generated.

Correspondingly, to fire a transition with label fcR (or nsR or fc or ns respectively) that starts in state si and ends in state sj for a token T=(nw/sj, nt/su) with start state sj in bottom-up direction means the following: to check, whether there exists a node with ID nv in the XML tree such that the node with ID nw is the first-child (or the next-sibling or fcR or previous-sibling respectively) of the node with ID nv. If such a node exists, a token T'=(nv/si, nt/su) is generated, otherwise no token is generated.

A transition with a label self from the start state si to the end state sj can be fired for each token having si as final state in case of top-down evaluation and for each token having sj as start state in case of bottom-up evaluation. Firing the transition during top-down evaluation generates for each token T=(nu/sz, nv/si) another token T'=(nu/sz, nv/sj), whereas firing the transition during bottom-up evaluation generates for each token T=(nv/sj, nu/sz) another token T'=(nv/si, nu/sz).

Whenever a token with a non-stable state s' is generated during sub-query evaluation in a direction D (bottom-up or top-down), all transitions that can be fired from s'

in the same direction as D are fired. Thereafter, tokens containing s' are deleted. This processing of tokens containing unstable states is repeated until all existing tokens have reached stable states again. When this happens, the DecisionModule gets the control again and decides which tokens are transferred next.

Consider for example the tokens (3/s6, 3/s6), (5/s6, 5/s6), and (7/s6, 7/s6) representing answers to the sub-query //self::b. If we fire the transitions from state s6 to state s3 in bottom-up direction, this removes and transfers the tokens having state s6 as their start state, and it will stop, when all tokens are either deleted or transferred to tokens having state s3 as their start state. In this case, the generated tokens (2/s3, 3/s6), (1/s3, 5/s6) say that the sub-query self::a/b represented by the sub-automaton between the states s3 and s6 applied to the XML nodes with IDs 2 (and 1 respectively) yields as answers the XML nodes that have the IDs 3 (and 5 respectively).

If we additionally create a token (0/s0, 0/s0) in the state s0 for the XML root node and transfer this token from s0 top-down, when token generation stops, we get the tokens (0/s0, 1/s3), (0/s0, 2/s3),(0/s0, 4/s3), and (0/s0, 6/s3) saying that the sub-query //a represented by the states of the sub-automaton between state s0 and state s3 returns the XML nodes with IDs 1, 2, 4, and 6 as answers.

Token Joining: Whenever at the end of a token transfer phase, the same stable state sy occurs as final state in tokens T1 and as start state in other tokens T2, we perform a so called 'token joining' and join those pairs (T1,T2) of tokens that relate sy to the same XML node nv with each other. A token joining of two tokens T1=(sx/nu, sy/nv) and T2=(sy/nv, sz/nw) yields a new join token T3=(sx/nu, sz/nw).

In our example, the tokens T1 ∈ { (2/s3, 3/s6) , (1/s3, 5/s6) } have the start state 3, and the tokens T2 ∈ { (0/s0, 1/s3), (0/s0, 2/s3),(0/s0, 4/s3), (0/s0, 6/s3) } have the final state 3. If we perform token joining on all pairs (T1,T2) of tokens, we get the join tokens j1=(0/s0, 3/s6) and j2=(0/s0, 5/s6). These join tokens express that the answers to the concatenated sub-query //a/self::a/b represented by the automaton between state s0 and state s6 applied to the XML node with ID 0 (the root node) returns the XML nodes with IDs 3 and 5 as answers. As all incoming and outgoing transitions of s3 have been fired, and all join tokens involving s3 have been computed, thereafter all tokens containing s3 as start state or as final state are deleted.

Token joining can also be used for finally joining the answers when query evaluation starts at an inner state and proceeds in different directions. If we declare for example state s3 as the single start state and transfer the tokens (1/s3, 1/s3), (2/s3, 2/s3), (4/s3, 4/s3), and (6/s3, 6/s3) top-down and bottom-up, 6 additional tokens are generated and the state s0 occurs as start state in the 4 tokens (0/s0, 1/s3), (0/s0, 2/s3), (0/s0, 4/s3), and (0/s0, 6/s3) and the state s6 occurs as final state in the 2 tokens (1/s3, 5/s6) and (2/s3, 3/s6). Finally, token joining calculates the final results (0/s0, 3/s6) and (0/s0, 5/s6) that express that by applying the query //a/b represented by the (sub-) automaton from the start state s0 to the final state s6 to the XML node with ID 0 (the root node) yields the XML nodes with IDs 3 and 5 as query results.

2.4 Optimization Using a Navigation Cache

If we consider the XML tree of Fig. 2(b) and the query //a//b and the nodes with ID 5 and with ID 7 and transfer the tokens bottom-up in a naïve way, similar new tokens are generated for the nodes a2, a1, and #root0, i.e., we pass this path in the tree more

than once. In order to overcome this weakness, we have introduced the concept of a so called *navigation cache* that caches tokens representing sub-query evaluations of multiple paths in the XML document tree.

For example, if the token for b5 is transferred first, the token for b7 can read the cache information of node b5 and can be transferred directly to the root node without having to pass the path via b5, a2, a1, and #root0 a second time. This information is being used for bottom-up evaluation only and is not considered for top-down evaluation.

2.5 Evaluation of Queries with Filters

Whenever a location step L that is represented by a pair (sx,sy) on the main path of an XPath query Q contains one or more predicate filters, each predicate filter Fi is represented by a filter automaton Ai having a state si as its root and a final state sfi representing the final state of the main path of the filter. A filter automaton Ai has the same design and functionality as the automaton for the main path of Q as described in the previous section. As with all location steps, each location step within a filter automaton can be evaluated top-down or bottom-up.

Token transfer between the root state si of a filter automaton Ai and the final state sy representing the location step L having filter Fi, can be done either top-down, i.e. from sy to si, if tokens containing sy are generated first, or bottom-up, i.e. from si to sy, if Fi is evaluated first.

Bottom-Up Token Transfer: If the tokens are transferred bottom-up, i.e. the filter path for Fi is evaluated before tokens containing the node sy are generated, let ST={ (n1/si, nf1/sf1), …, (nk/si, nfk/sfk) } be the set of all the tokens computed for path from si to sfi. Then the set N={n1,…nk} contains exactly those XML nodes for which Fi evaluates to true.

Then, the automaton state sy to which the filter path is connected reacts similar to a state without an attached filter with the difference, that not for each XML node that fulfills the given node name test a token is created, but only for those XML nodes contained in the set N which fulfill the given node name test.

Top-Down Token Transfer: Otherwise, i.e., if tokens are transferred top-down from sy to si, we follow an idea of [10]: Whenever a token T1=(…,nv/sy) or a token T1= (nv/sy,…) that contains the state sy is generated, this token gets a reservation that depends on whether or not the filter automaton for Fi evaluates to `true` for the XML node nv. At the same time, the filter automaton for Fi is switched active, i.e., a token (nv/si, nv/si) is generated which turns si into a start state of the filter automaton.

If the filter automaton Fi finally evaluates to `true` for the XML node nv, the reservation for Fi is deleted. We say that the execution of the filter automaton for Fi having a start state si and a final state sfi on its main path evaluates to `true` for the XML node nv, if and only if eventually a token (nv/si, nw/sfi) is generated for a XML node nw. Otherwise, we say that the evaluation of the filter automaton for Fi evaluates to `false` for the XML node nv, and the token T1 itself is deleted and considered invalid. However, if, finally, all the reservations for a filter attached to sy are deleted, the T1 token is considered valid.

The states of the filter automaton can be connected to other filter automata, such that nested filter automata for implementing nested XPath filter expressions can be evaluated by this concept as well.

2.6 Evaluation Strategies

We have implemented different types of DecisionModules that follow different evaluation strategies in order to evaluate an XPath query. The first two Decision Modules follow the 'traditional' ways to evaluate queries.

- The Top-Down-Module declares the root state as the only start state. Tokens are added to the first state of each filter automaton FA as soon as a token is added to the state to which FA is attached to. All paths, i.e. the main path of the XPath expression and all filter paths, are evaluated top-down.
- The Bottom-Up-Module declares the target state of the automaton for the main path and each target state of a filter automaton as the start states. All paths are evaluated bottom-up.
- The Minimum-Module considers the locations steps in the main path and in all filter paths and declares the stable state of that location step having the lowest selectivity of the whole query as the only start state. If the start state is part of a filter, the corresponding filter path is evaluated top-down and bottom-up starting at the start state and the result is added to the state of the main path to which the filter is attached. Then, the state of the main path behaves like a start state: From that given start state, the remaining main path of the XPath query is evaluated bottom-up to the root and top-down to the target state of the main path. Furthermore, all other filter paths are evaluated top-down.

Determining the location step having the lowest selectivity is not trivial. Currently, we are using a simple heuristics that regards that location step LS=/axis::nnt as the location step having the lowest selectivity, for which the least number of nodes exist in the document that fulfill the node name test nnt.

3 Evaluation of Our Prototype Implementation

3.1 Experimental Setup

Our test system has an Intel Core 2 Duo with 2,53 GHz (T9400) processor and 4 GB 1066 DDR 3 RAM. The prototype is implemented in Java and runs on JDK 1.6 Update 21 with an extended RAM and function stack (parameters -Xmx1300M -Xss4096k). For MonetDB [11], we have used the Oct2010-SP1 build and the measured execution time is Trans+Shred+Query. For eXist-DB (http://exist-db.org/) we have used version 1.4.0.

Our evaluation was performed on the documents generated by the XMark benchmark [12] with original XML document sizes varying from around 2 MB to 50 MB. We have evaluated our prototype on the queries A1-A7 and B2-B4 of the XPathMark [13] benchmark suite as well as on some additional, practice-oriented queries (Q1-Q6) for showing the advantages of our system (especially on queries with location steps of outstanding low selectivity). The queries that we used are shown in Table 2.

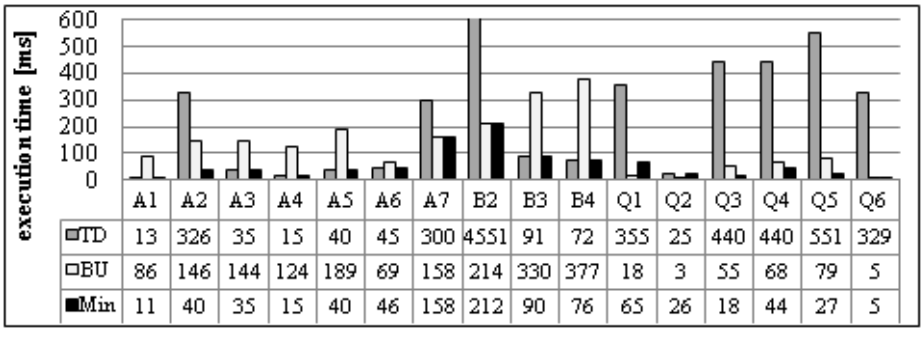
Table 2. Queries used in our prototype evaluation

A1	/site/closed_auctions/closed_auction/annotation/description/text/keyword
A2	//closed_auction//keyword
A3	/site/closed_auctions/closed_auction//keyword
A4	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date
A5	/site/closed_auctions/closed_auction[descendant::keyword]/date
A6	/site/people/person[profile/gender and profile/age]/name
A7	//keyword
B2	//keyword/ancestor::listitem/text/keyword
B3	/site/open_auctions/open_auction/bidder[following-sibling::bidder]
B4	/site/open_auctions/open_auction/bidder[preceding-sibling::bidder]
Q1	//people//age
Q2	/site/people/person[profile/age=42]
Q3	//person[.//gender='female']/name
Q4	//person[.//country='United States']/name
Q5	//person[.//country='United States' and .//gender='female']/name
Q6	//item[payment='Creditcard']

3.2 Comparison of DecisionModules

In our first series of measurements, we compared the three DecisionModules with each other. We performed all measurements on two XML representations: on succinct compression as compressed XML representation and on DOM as uncompressed XML representation. As the experiments have shown that using our prototype based on the Java DOM representation yields similar execution times, but requires 5 times more main memory than the execution based on the succinct compression, we concentrate on presenting the results received for the succinct compression in this section.

Fig. 3 compares the three DecisionModules "Top-Down(TD)", "Bottom-Up(BU)" and "Minimum(Min)". Navigation Caching is always enabled, as our evaluations have shown that this technique in general improves the performance. The evaluation shown in Fig. 3 was performed on a ~22MB document with 340,000 nodes (XMark factor 0.2), but other document sizes show the same results as the execution times scales linear with increasing document size. The overall observation is that strategy BU outperforms the execution time of the strategy TD in most cases, but that Min is the best evaluation strategy in nearly all cases (except for query Q1 and Q2).



	A1	A2	A3	A4	A5	A6	A7	B2	B3	B4	Q1	Q2	Q3	Q4	Q5	Q6
TD	13	326	35	15	40	45	300	4551	91	72	355	25	440	440	551	329
BU	86	146	144	124	189	69	158	214	330	377	18	3	55	68	79	5
Min	11	40	35	15	40	46	158	212	90	76	65	26	18	44	27	5

Fig. 3. Comparison of the DecisionModules TD, BU, and Min

An example for this observation is query A2: A top-down evaluation traverses the entire document due to the initial descendant axis. The bottom-up module can do better because it starts at the *keyword* nodes and evaluates the descendant location steps bottom-up. But the Min-Module performs best on A2: It has a low selectivity, i.e., it selects the 2,000 *closed_auction* nodes (compared with 14,000 *keyword* nodes) as start nodes and therefore can avoid a lot of the navigation caching overhead of the BU-Module, but only has to traverse relative small sub-trees (the *closed_aution* sub-trees) compared with the entire document the TD-Module has to traverse. A similar behavior can be observed on query A7.

In some of the XPathMark-queries (A1, A3, A4, A5, A6), the Min-Module behaves similar to the TD-Module. As these queries do not have descendant-axis location steps or the descendant-axis steps are at the end of query, top-down evaluation is nearly optimal because only relative small sub-trees are traversed.

We have added Q1-Q6 to show results of more complex queries. On query Q4, the Min-Module can profit from the filter, as only a fraction of the people are from the *United States* (around 38%). Therefore, the Min-Module starts at all *United States* text-nodes, evaluates the *person* descendant-axis location step bottom-up and the *name* location step top-down. This evaluation strategy improves the performance significantly: the Min-Module is 10 times faster than the TD-Module and 1.5 times faster than the BU-Module.

Note that the Min-Module can sometimes perform better on *more complex* queries if selectivity becomes lower: Query Q5 is evaluated faster than query Q4, because the additional filter condition *gender='female'* further reduces the number of selected *person* nodes (only 5% of the *persons* are *female* and from the *United States*). In this case, the Min-Module is 20 times faster than the TD-Module and 3 times faster than the BU-Module.

3.3 Comparison with Other Evaluators

In Fig. 4, we measured the average execution time of all queries (A1-A7,B2-B4, Q1-Q6). As we can observe, the Module "Min" of our prototype is scaling linear with increasing document size. Furthermore, it is significantly outperforming JAXP, as it is around 15 times faster than JAXP. The Module "Min" outperforms eXist-DB as well. For files up

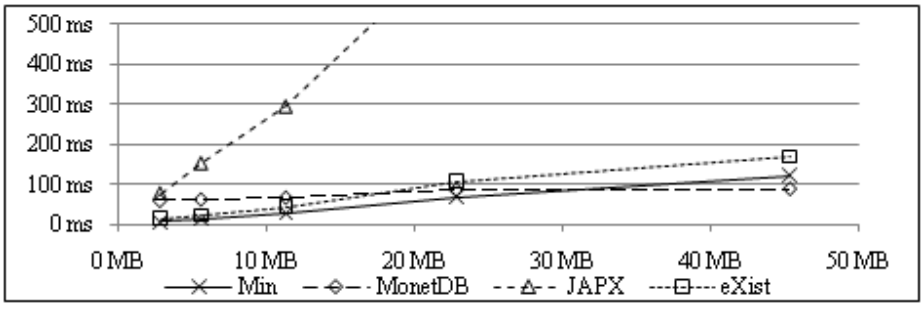


Fig. 4. Comparison of our prototype with other XPath evaluators

to ~30 MB, the Module "Min" is outperforming MonetDB as well. This is due to the fact that MonetDB needs a high overhead for query optimization but scales nearly constantly for for the document sizes tested in our evaluation. For queries with low selectivity (e.g. Q3 and Q6) our approach can outperform MonetDB also for files having a size of more than 30 MB. Furthermore, when comparing run-time, note that our prototype is a Java application, whereas MonetDB is a strongly optimized C application.

4 Related Works

There exist several different approaches to the evaluation of XPath queries on XML data. They can be divided into categories by the subset of XPath that they support. Nearly all of them are based on automata (X-scan[7], XMLTK[3], YFilter[6], [10],[13], [14], AFilter [5], XSQ [9], SPEX [8]) or parse trees ([15], [4], [16], [17]). All of them support the axes child and descendant-or-self and most of them support predicate filters and wildcards, but besides [10] and [18] none of them support the sibling-axes as our solution does.

The approach presented in [1] defines bottom-up as well as top-down semantics and presents an bottom-up and a top-down processing algorithm that both run in low-degree polynomial time for full XPath and an enhanced algorithm that runs in linear time for Core XPath that evaluates the main path top-down and the filter paths bottom-up. In contrast to this approach, we try to combine the advantages of bottom-up and top-down processing by choosing bottom-up or top-down evaluation for each location-step, such that an algorithm is developed that runs very efficient in practice. As our evaluation has shown, the mixed strategy MinimumModule performs and scales better that the pure strategies top-down or bottom-up.

For the automata-based approaches, the XML input stream is the controlling instance that is used as input for the automata representing the Query.[19] and [20] present a compressed representation for XML together with an XPath evaluator that is based on tree automata and that allows to skip irrelevant parts of the compressed XML document during the evaluation process. They allow selecting a single start point and follow the path to the root bottom-up and the path to the "leafs" of the query top-down. In contrast to[19] and [20], we allow the selection of any number of start points and the evaluation of the sub-queries in any direction.

The approach presented in[18] supports the axes self, child, descendant, following and following-sibling but does not support backward axes. It translates the queries into expressions over the binary axes first-child and next-sibling and then constructs a two-layered NFA that consumes the SAX events start-element, end-element and character. The first layer evaluates the main path of the query, whereas the second layer is responsible for the evaluation of the predicate filters. Our previous approach [10] supports all forward axes but supports backward axes only if they are rewritten to forward axes before query evaluation starts. It translates queries into an automaton that consumes the binary events first-child and next-sibling. It can evaluate streams in top-down direction only. XMLTK[3], and YFilter[6], [13], [14] and X-scan[7] are based on the lazy construction of deterministic finite automata (DFA), i.e., the DFA is not generated completely at the beginning, but additional states are added only when needed. AFilter [5] is adaptable in terms of the memory requirement, i.e., it needs a base memory 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. XSQ [9] and SPEX [8] 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.

Parse trees – in contrast to automata – take the control of the evaluation process themselves, i.e., they decide which node of the parse tree will be processed next and check with the XML input document, whether this node can be processed. The approach presented in [21] translates the input query into a set of parse trees. Whenever a matching of a leaf node 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. χαοζ[4] and [15] build a parse tree as well (plus a parse-dag in [4], as they support the parent and the ancestor axis in addition). This parse tree is used for 'predicting' the next matching nodes and the level in which they have to occur. The approach discussed in [16] collapses the parse tree into a prefix trie by combining common prefix sequences of child-axis location steps of different queries into a leaner single path of the prefix trie. The approach presented in [17] uses a parse tree that stores XML nodes that are solutions to the parse tree node's sub-query within a stack that is attached to each node.

The authors of [22] show that queries containing joins on attribute values can be computed in time linear of the XML document but exponentially of the query size. They evaluate one path to the join attribute top-down and the path to the second join attribute bottom-up. They require a special index on the attribute values and a pointer structure representation of the XML document, such that the idea is not applicable to arbitrary XML representations as e.g. compressed XML.

ROX [23] is a run-time optimizer for XQuery that is used as a MonetDB extension. It is based on an indexed representation of the XML document that is stored in form of relational data. It consists of a relational query optimizer for the 'relational parts' of an XQuery and an XML query optimizer that is intertwined with the query execution, i.e., that adapts the query execution plan during the query execution. In contrast to our approach, ROX can be applied to the indexed XML document in form of a relational representation only and cannot be applied to compressed XML.

In comparison to all these approaches, we additionally support the 'sibling'-axes following and following-sibling. Furthermore, beyond [21] and [9], 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. In comparison to [10] and [18], we have used an extended automata model which supports also bottom-up evaluation and mixed evaluation strategies.

5 Summary and Conclusions

Whenever XPath query evaluation is the bottleneck of an application, and main memory is small in comparison to memory requirements for fast query evaluation, a fast in-memory XPath evaluator that works also on compressed XML structures may be a significant improvement towards a better run-time.

In this paper, we have presented an XPath query processor that can evaluate XPath queries on each XML representation that supports a small number of basic binary

axes (first-child, first-child^{-1}, next-sibling, next-sibling^{-1}, and self), like e.g. DOM or the compressed XML representation 'Succinct' [2]. Our query processor decomposes and normalizes each XPath query, such that the resulting path queries contain only the basic binary axes, and then converts them into lean token automata. A DecisionModule decides for each location step which evaluation strategy to follow, i.e., which location step to evaluate when and in which direction.

Our tests have shown, that our query processor is very efficient and outperforms other approaches like JAXP provided by JDK 1.6 and yields results faster than MonetDB – a database that allows the native storage of XML files and that uses an index on this data to speed up the query evaluation – for files up to ~30 MB in general or for queries with at least one location step that has a low selectivity.

As XPath is being 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. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. ACM Trans. Database Syst. 30, 444–491 (2005)
2. Böttcher, S., Hartel, R., Heinzemann, C.: BSBC: Towards a Succinct Data Format for XML Streams. In: WEBIST 2008, Funchal, Madeira, Portugal, pp.13–21 (2008)
3. Avila-Campillo, I., Green, T., Gupta, A., Onizuka, M., Raven, D., Suciu, D.: XMLTK: An XML toolkit for scalable XML stream processing. In: Proceedings of PLANX (2002)
4. Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M., Josifovski, V.: Streaming XPath Processing with Forward and Backward Axes. In: ICDE, Bangalore, India, pp. 455–466 (2003)
5. Candan, K., Hsiung, W.-P., Chen, S., Tatemura, J., Agrawal, D.: AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering. In: VLDB, Seoul, Korea (2006)
6. Diao, Y., Rizvi, S., Franklin, M.: Towards an Internet-Scale XML Dissemination Service. In: VLDB, Toronto, Canada, pp. 612–623 (2004)
7. Ives, Z., Halevy, A., Weld, D.: An XML query engine for network-bound data. The VLDB Journal 11(1), 380–402 (2002)
8. Olteanu, D., Kiesling, T., Bry, F.: An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. In: ICDE, Bangalore, India, pp. 702–704 (2003)
9. Peng, F., Chawathe, S.: XPath Queries on Streaming Data. In: ACM SIGMOD, San Diego, California, USA, pp.431–442 (2003)
10. Böttcher, S., Steinmetz, R.: Evaluating XPath Queries on XML Data Streams. In: Cooper, R., Kennedy, J. (eds.) BNCOD 2007. LNCS, vol. 4587, pp. 101–113. Springer, Heidelberg (2007)
11. Boncz, P., Grust, T., Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery: a fast XQuery processor powered by a relational engine (2006)
12. Schmidt, A., Waas, F., Kersten, M., Carey, M., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In: VLDB, Hong Kong, pp.974–985 (2002)
13. Green, T., Gupta, A., Miklau, G., Onizuka, M., Suciu, D.: Processing XML streams with deterministic automata and stream indexes. ACM Trans. Database Syst. 29 (2004)
14. Gupta, A., Suciu, D.: Stream Processing of XPath Queries with Predicates. In: ACM SIGMOD, San Diego, California, USA, pp.419–430 (2003)

15. Bar-Yossef, Z., Fontoura, M., Josifovski, V.: On the memory requirements of XPath evaluation over XML streams. J. Comput. Syst. Sci. 73(3), 391–441 (2007)
16. Chan, C., Felber, P., Garofalakis, M., Rastogi, R.: Efficient Filtering of XML Documents with XPath Expressions. In: ICDE, San Jose, CA, USA, pp.235–244 (2002)
17. Chen, Y., Davidson, S., Zheng, Y.: An Efficient XPath Query Processor for XML Streams. In: ICDE 2006, Atlanta, GA, USA, p.79 (2006)
18. Onizuka, M.: Processing XPath queries with forward and downward axes over XML streams. In: EDBT 2010, Lausanne, Switzerland, pp.27–38 (2010)
19. Arroyuelo, D., Claude, F., Maneth, S., Mäkinen, V., Navarro, G., Nguyen, K., Siren, J., Välimäki, N.: Fast in-memory XPath search using compressed indexes. In: ICDE 2010, Long Beach, California, USA, pp. 417–428 (2010)
20. Maneth, S., Nguyen, K.: XPath Whole Query Optimization. PVLDB 3(1), 882–893 (2010)
21. Josifovski, V., Fontoura, M., Barta, A.: Querying XML streams. VLDB Journal 14, 197–210 (2005)
22. Bojanczyk, M., Parys, P.: XPath evaluation in linear time, pp. 241–250 (2008)
23. Kader, R., Boncz, P., Manegold, S., Keulen, M.: ROX: run-time optimization of XQueries, pp. 615–626 (2009)