XPathMark
Functional and Performance Tests for XPath

M. Franceschet

Department of Mathematics and Computer Science
University of Udine

Abstract. We present a major revision of the XPath benchmark known as XPathMark [1]. The new version splits into a functional test over a small educational document and a more elaborated performance test over XMark [2] documents. We conclude by sharing with the reader our experience on running XPathMark on some popular XSLT/XQuery processors.

1 Functional and Performance Tests for XPath

The new release of XPathMark [3] consists of a functional and performance test for the XPath language version 1.0.

1.1 XPath Functional Test

The main goal of *XPath Functional Test* (*XPath-FT*) is testing *completeness* (which features of the language are supported?) and *correctness* (which features of the language are correctly implemented?) of an XML query processing system with respect to XPath 1.0.

XPath-FT contains several groups of queries each covering a different *functional* aspect of the language including navigational axes, filters, node tests, operators and functions. The queries are interpreted over a small educational document and each query is accompanied with the correct answer. The skeleton of the target XML document, depicted in Figure 1, rapresents the English alphabet in such a way that the preorder traversal of the XML tree corresponds to the English alphabet sorted from A to Z. As an example of the query set, we report four queries in the axes group along with their answers in graphical format (see red nodes in Figure 2):

A3 *The descendant nodes of L* (Answer: Figure 2 top-left)

```
//L/descendant::*
```

A5 *The ancestor nodes of L* (Answer: Figure 2 top-right)

```
//L/ancestor::*
```

A9 *The following nodes of L* (Answer: Figure 2 bottom-left)

Dagstuhl Seminar Proceedings 06472
XQuery Implementation Paradigms
http://drops.dagstuhl.de/opus/volltexte/2007/892

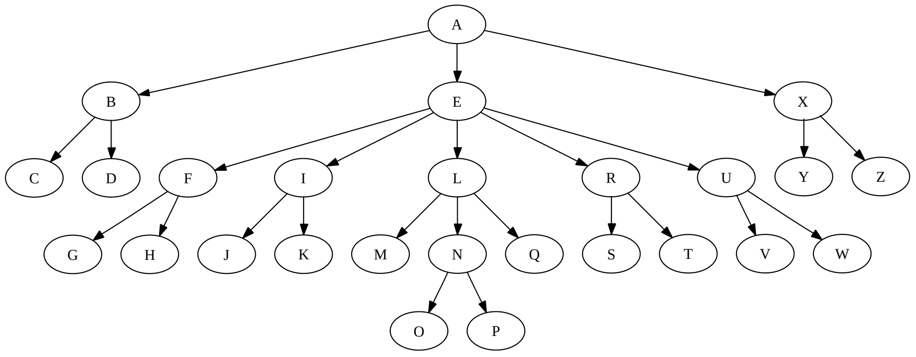

Fig. 1. The alphabet tree

`//L/following::*`

A10 *The preceding nodes of L* (Answer: Figure 2 bottom-right)

`//L/preceding::*`

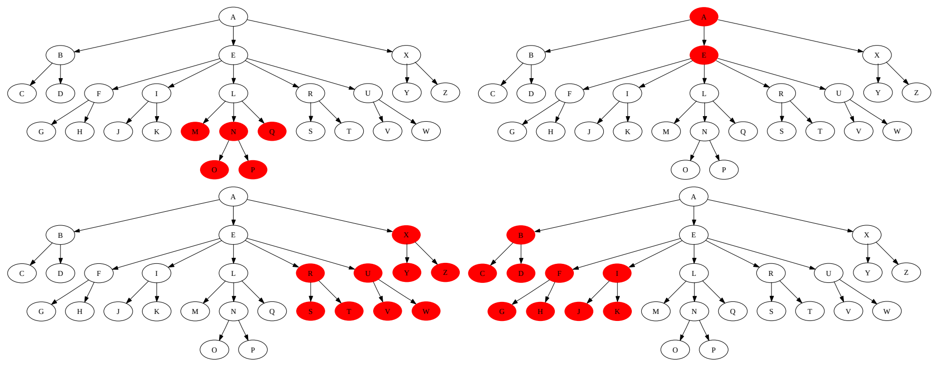

Fig. 2. Query answers (red nodes)

The full list of queries along with their answers (in textual format and, when possible, in graphical format also) is accessible at the XPathMark website [3]. Incidentally, XPath-FT can also be used as an educational tool to learn XPath, possibly accompanied with a XPath visualizer like BaseX (`http://www.inf.uni-konstanz.de/dbis/research/basex`).

1.2 XPath Performance Test

XPath Performance Test (*XPath-PT*) aims at investigating the performance of an XML query processor with respect to XPath 1.0 in terms of time spent

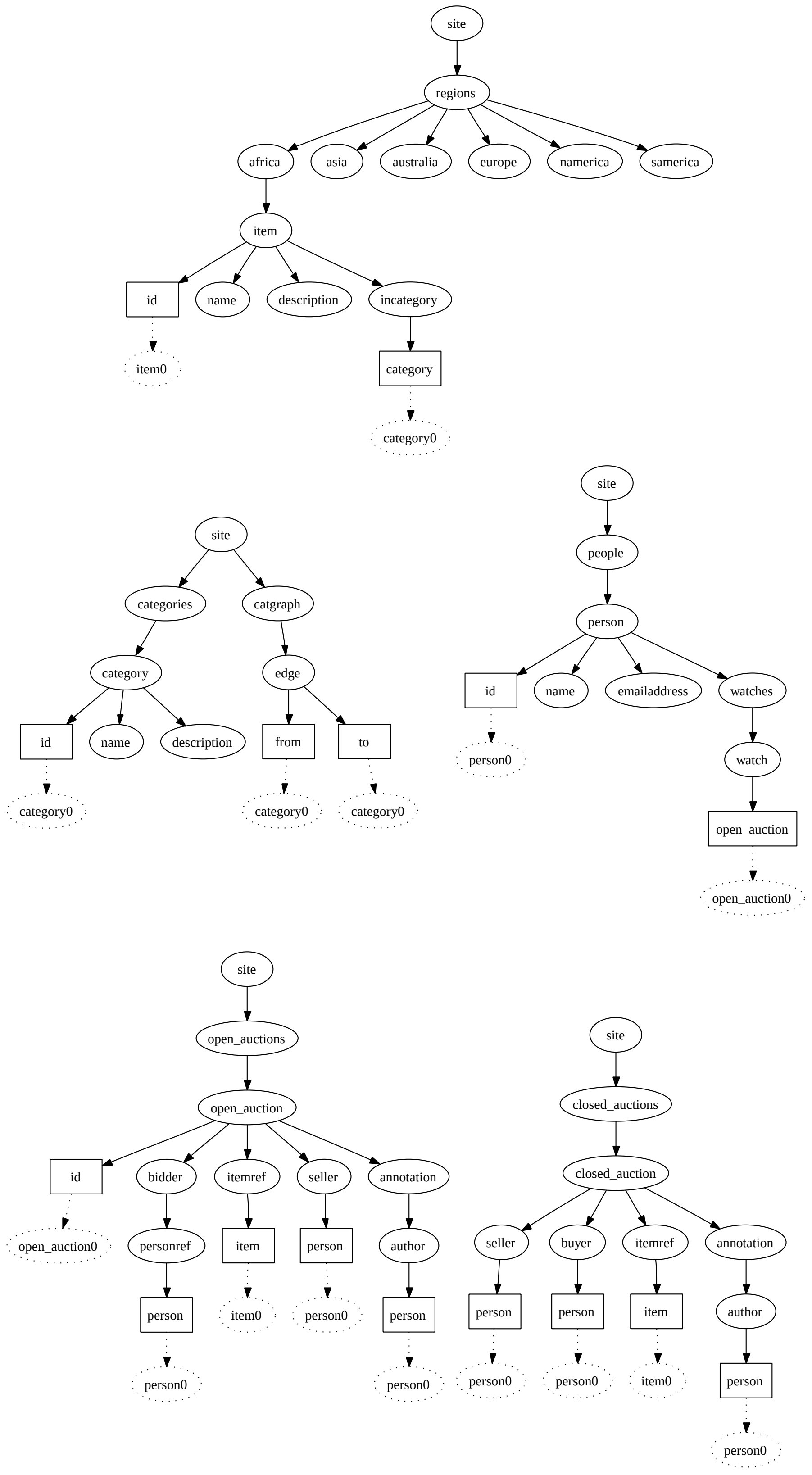

Fig. 3. XMark document components

to execute a query. Queries of the test are divided into groups according to the *intrinsic computational complexity* of the corresponding evaluation problem. They are defined to challenge both data and query scalability and include some instances of transitive closure of path expressions. In the following we highlight the main features of XPath-PT:

1. queries simulate realistic query needs of a potential user of an auction site. They have a natural interpretation over documents generated with the popular XML benchmark *XMark* [2]. XMark provides an efficient and easy to use data generator. It generates scalable documents modeling an Internet auction website with different components like bidders, items, interest categories, open and closed auctions. The documents contain both data- and text-centric XML fragments. A simplified version of the structure of the different components of an XMark document is depicted in Figure 3;
2. queries are divided into groups according to the *intrinsic computational complexity* of the corresponding evaluation problem. XPath language can be stratified in a number of fragments for which different complexity bounds are known [4]. Comparing the theoretical computational complexity of the query evaluation problem with the actual amount of resources consumed during query evaluation might be, at least, a stimulating and instructive exercise;
3. queries are defined to challenge *data scalability* of the XML processing system, that is the performance of the system as the data complexity (document size) grows. In particular, the designed queries talk about document sections (like open and closed auctions, items, people, descriptions) that become bigger when the XMark document scaling factor increases. Moreover, the results of the queries are small compared to the size of the target document. This avoids that the time taken to serialize the results (that may be relevant) obfuscates the pure query processing time;
4. some of the queries are defined to challenge *query scalability* of the XML processing system, that is the performance of the system as the query complexity grows. These queries are parametric with respect to a given factor that influences the query complexity, like length or nesting degree of predicates, and for each parametric query a generator is provided;
5. finally, the test includes some instances of queries using *transitive closure* of path expressions. A typical example of query using transitive closure is reachability: find all nodes that are reachable from a given node trough an arbitrary path in a graph. Transitive closure of path expressions is beyond the expressive power of XPath. The addition of transitive closure to XPath has been investigated from a theoretical point of view in [5, 6]. However, a practical counterpart consisting of different approaches, implementations, and optimizations to tackle the intrinsic complexity of transitive closure in XPath is still missing, and a benchmark might help to start this endaveour.

XPath-PT consists of six groups of queries. Each group contains queries belonging to a particular XPath language. The six XPath languages form an inclusion chain in terms of expressivity and complexity. In the following we describe the XPath languages that we consider and give corresponding examples taken from XPath-PT. The full list of queries, along with generators for parametric ones, can be found at the XPathMark website [3].

XPath-A

This fragment contains so-called *unary tree pattern* queries. These queries use only child and descendant axes, node tests equal to * or to a tag name, and filters (predicates). Conjunctive and disjunctive Boolean operators are allowed, but negation is not. Relational and arithmetic operators and functions are disallowed. Some examples follow:

A4 *Closed auctions with an annotation containing a keyword*

```
/site/closed_auctions/closed_auction
[annotation/description/text/keyword]
```

A6 *People that have declared both gender and age*

```
/site/people/person[profile/gender and profile/age]/name
```

XPath-B

This fragment contains so-called *core* or *navigational* XPath queries. This fragment extends XPath-A by allowing all XPath axes and negation. Some examples follow:

B3 *Bidders except the last one of each open auction*

```
/site/open_auctions/open_auction/bidder[following-sibling::bidder]
```

B8 *Open auctions with exactly one bidder*

```
/site/open_auctions/open_auction[bidder and
not(bidder/preceding-sibling::bidder)]
```

B13(i) *Parametric query on query length (meant to challenge query scalability)*

`//keyword(/ancestor::parlist/descendant::keyword)`^{i} for $i \geq 0$, where `path`^{i} means `path` repeated i times.

XPath-C

This fragment contains *relational* XPath queries. This fragment extends XPath-B by allowing all relational operators (`=`, `!=`, `>`, `<`, `>=`, `<=`) and the `id()` function. Some examples follow:

C3 *People with an income equal to the current price of some item*

```
/site/people/person[profile/@income =
/site/open_auctions/open_auction/current]/name
```

C4 *People that are sellers in an auction that they are watching*

```
/site/people/person
[watches/watch/id(@open_auction)/seller/@person = @id]/name
```

C8(i) *Categories that are reachable from a given category in i steps in the category graph, for $i \geq 1$ (parametric query on nesting degree of filters. It is meant to challenge query scalability.)*

`id(Y(i))/name` for $i \geq 1$, where:

`Y(1) = /site/catgraph/edge[@from = "category0"]/@to`
`Y(i) = /site/catgraph/edge[@from = Y(i-1)]/@to` for $i \geq 2$

XPath-D
This fragment contains *arithmetic* XPath queries. This fragment extends XPath-C by allowing all arithmetic operators (`+`, `-`, `*`, `div`, `mod`) and functions `sum()` and `count()`. Some examples follow:

D2 *The number of pieces of prose contained in the document*

```
count(//text) + count(//bold) + count(//emph) + count(//keyword)
```

D5 *Open auctions with an average increase greater than the double of the initial price*

```
/site/open_auctions/open_auction[bidder and
(sum(bidder/increase) div count(bidder)) > 2 * initial]
```

XPath-E
This fragment contains all XPath 1.0 queries. In particular, it extends XPath-D by allowing all functions (like `position()` and `contains()`). Some examples follow:

E1 *Open auctions whose increase in the median position is contained between the first and the last increase of the auction*

```
site/open_auctions/open_auction[number(bidder[1]/increase) <
number(bidder[floor((last() + 1) div 2)]/increase) and
number(bidder[floor((last() + 1) div 2)]/increase) <
number(bidder[last()]/increase)]
```

E5 *Items that have at least 100 items following them in the document and at least 100 items preceding them in the document*

```
/site/regions/*/item[preceding::item[100] and
following::item[100]]/name
```

E6 *Items whose description contains the name of the item*

```
/site/regions/*/item[contains(description, name)]/name
```

E7 *Items whose description contains the word* passion *followed by the word* eros *followed by the word* dangerous

```
/site/regions/*/item[
contains(substring-before(description, "eros"), "passion") and
contains(substring-after(description, "eros"), "dangerous")]/name
```

XPath-F

This fragment contains all *regular* XPath queries. It extends XPath-E with `idref()` and `closure()` functions. While `idref()`, the natural counterpart of `id()`, is defined in XPath 2.0, `closure()` is new to XPath and XQuery (though it has been added to EXSLT (`http://www.exslt.org`), an extension of XSLT). The closure function takes two arguments: `closure(C, path)`, where `C` is a node set and `path` is a location path in XPath 1.0 (an expression that computes a node set). The closure function computes the transitive closure of `path` starting at `C`, that is, the set of nodes that are reached in one or more applications of `path` setting `C` as the initial context set (see Section 1.3 for the precise definition). For instance, `closure(/, child::*)` returns all descendants of the root, while `closure(/, child::a)` returns all descendants of the root that can be reached trough a path labelled with `a`. Some examples follow:

F1 *Bidders such that there exists a following sibling bidder with increase bigger than 10 and for all bidders in between the increase is smaller or equal than 10*

`//bidder[number(increase) <= 10 and (BIG or closure(.,SMALL)/BIG)]`, where

```
BIG = following-sibling::bidder[position()=1 and number(increase)
> 10]
SMALL = following-sibling::bidder[position()=1 and number(increase)
<= 10]
```

F3 *Paragraph items that contain a keyword nested under an odd number of paragraph items*

```
//listitem[text/keyword or
closure(.,parlist/listitem/parlist/listitem)/text/keyword]
```

F8 *Categories that are reachable from a given category through an arbitrary path in the category graph*

```
id(//category[@id="category0"]/@id/
closure(.,idref(.)[name() = "from"]/../@to))/name
```

1.3 A Note on Transitive Closure

In this section we formally define the notion of transitive closure and describe an algorithm to compute it. Let D be a finite domain of cardinality n and $\Phi : 2^D \rightarrow 2^D$ be a function from the power set of D to the power set of D. We assume that Φ satisfies the following *decomposition property*. For any subset X of D:

$$\Phi(X) = \bigcup_{x \in X} \Phi(\{x\})$$

We define $\Phi^0(X) = X$ and, for $i \geq 1$, $\Phi^{i+1}(X) = \Phi(\Phi^i(X))$. The *transitive closure* of Φ is the function $\Phi^+ : 2^D \to 2^D$ such that:

$$\Phi^+(X) = \bigcup_{i=1}^{\infty} \Phi^i(X)$$

The question is: given an algorithm to compute Φ, what is an algorithm to compute the transitive closure Φ^+ of Φ? Before giving an answer to this question, let's link it to XPath. A location path π in XPath can be interpreted as a function Φ_π that takes a set of nodes X as input and outputs the result of the evaluation of π at X. Notice that, by definition of the semantics of XPath [8], Φ_π satisfies the decomposition property stated above. The result of the closure function $closure(X, \pi)$ is in fact the value of $\Phi^+_\pi(X)$.

Let us define

$$\Phi_i(X) = \bigcup_{j=1}^{i} \Phi^j(X)$$

The following two trivial observations hold for any Φ (even without assuming the decomposition property):

(OBS1) There exist $1 \leq i < k \leq 2^n$ such that $\Phi^k(X) = \Phi^i(X)$.

That is, after at most 2^n steps (the number of distinct subsets of the domain D), Φ generates a result that was already generated in the past. It follows that there is $k \leq 2^n$ such that $\Phi^+(X) = \Phi_k(X)$. This induces an (expensive) algorithm to compute $\Phi^+(X)$ in the general case. Moreover,

(OBS2) There exists $2 \leq k \leq n$ such that $\Phi^k(X)$ is contained in $\Phi_{k-1}(X)$.

That is, after at most n steps (the number of distinct elements of the domain D), Φ generates a result that does not contains any new element. The worst-case ($k = n$) happens when X is a singleton and each iteration of Φ adds a different element.

Exploiting the decomposition property of Φ, it is not difficult to show the following two observations:

(OBS3) If, for some $k \geq 2$, it holds that $\Phi^k(X)$ is contained in $\Phi_{k-1}(X)$, then, for each $i > k$, it holds that $\Phi^i(X)$ is also contained in $\Phi_{k-1}(X)$.

In other words, if $\Phi^k(X)$ does not produce any new element, then no further application of Φ will produce any new element. Putting together (OBS2) and (OBS3) it follows that there is $k \leq n$ such that $\Phi^+(X) = \Phi_k(X)$. This induces an algorithm to computes $\Phi^+(X)$ that terminates in at most n applications of Φ (assuming the decomposition property for Φ). Furthermore:

(OBS4) For $i \geq 1$ it holds that:

$$\Phi_i(X) = \Phi(X) \cup \bigcup_{j=2}^{i} \Phi(\Phi^{j-1}(X)) = \Phi(X) \cup \bigcup_{j=2}^{i} \Phi(\Phi^{j-1}(X) \setminus \Phi_{j-2}(X))$$

It follows that we can compute each iteration of Φ only on the new elements, that is on the elements that have been discovered during the last iteration. This induces an optimization in the previous algorithm since, instead of computing $\Phi^j(X)$ as the application of Φ to the last result $\Phi^{j-1}(X)$, we can compute it as an application of Φ to the smaller set $\Phi^{j-1}(X) \setminus \Phi_{j-2}(X)$.

The induced optimized algorithm to compute the transitive closure of the function Φ can be encoded as follows:

```
closure($\mathbf{X}, \mathbf{\Phi}$)
$result := \emptyset$
$new := X$
while $new \neq \emptyset$ do
  $current \leftarrow \Phi(new)$
  $new \leftarrow current \setminus result$
  $result \leftarrow result \cup new$
end while
return $result$
```

If the cost of computing Φ is C, then the worst-case complexity of the above algorithm is $O(n \cdot C)$. For instance, we know that it is possible to evaluate Core XPath (XPath-B) queries in time $O(n \cdot k)$, with n the data complexity and k the query complexity. It follows that queries in Core XPath plus transitive closure can be solved in time $O(n^2 \cdot k)$ in the worst-case.

It is worth noticing that the above algorithm for transitive closure can be easily encoded in a user-defined recursive function in XQuery (see XPathMark website [3] for an encoding of such a function). In this way it is possible to exploit an XQuery processor to compute transitive closure in XPath (and hence to run XPath-F fragment of XPath-PT).

2 Testing Experience

In this section we would like to fully report our testing experience with XPath-Mark. Most of experimental papers only report the last part of their testing experience consisting of some of the experimental results and possibly their interpretation. This is done mostly for space reasons. However, we are aware that testing takes a large portion (sometimes the majority) of the time spent to produce the whole paper. Moreover, the technical details involved in testing might be relevant to other researchers in order to reproduce and extend the experiments, or to perform similar testing. For these reasons, we decided to present in this section our full testing experience with XPathMark, from setting of the experimental framework to interpretation of the experimental results.

We ran both XPath Functional and Performance Tests on the following XML query processors:

1. XSLTproc (`http://xmlsoft.org/XSLT`) version 10114, an XSLT processor developed in C for the Gnome project.

2. Xalan-Java (`http://xml.apache.org`) version 2.7.0, an XSLT processor witten in Java from The Apache Software Foundation.
3. SaxonB (`http://saxon.sourceforge.net`) version 8.8J, an XSLT and XQuery processor written in Java from Saxonica of Michael Kay.
4. Qizx/open (`http://www.axyana.com/qizxopen`) version 1.1, an XQuery processor witten in Java from Axyana Software.

All the tested processors support XPath 1.0 and are freely available on the Internet. We ran the tests on an Intel Pentium III with CPU at 1 GHz, 256 MB of RAM, running Linux version 2.6.11-1.1369_FC4. We took advantage of XCheck [9], a benchmarking platform for XML query engines. This saved much time by automatizing a lot of tedious and repetitive tasks. If you want to run yourself the tests or extend them, you may download XCheck-FT and XCheck-PT archives from XPathMark website, containing XPath-FT and XPath-PT tests in XCheck format. They include processor adapters, XML documents, queries, and the XCheck experiment document that can be integrated with XCheck software to automatically run the tests. You can easily extend the tests to other processors, documents, and queries.

It is well noticing that our goal here is not to show that one of the above engine is better than the other (even though, from our investigation, the reader can draw his or her own conclusions). The aim is to give an instance of how the proposed tests for XPath might be used to better understand the functional and performance behavior of XML query processors.

2.1 Experimenting with XPath Functional Test

We report here about our outcomes for XPath Functional Test. For Xalan and XSLTproc we prepared an XSLT version of each query as follows:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <answer><xsl:copy-of select="query/is/here"/></answer>
  </xsl:template>
</xsl:stylesheet>
```

For SaxonB and Qizx/open we prepared an XQuery version of each query as follows:

```
let $x := doc("alphabet.xml")/query/is/here
return <answer>{$x}</answer>
```

Moreover, we ran XCheck with the following command:

```
$ ./XCheck.pl -n 1 -s -run xpath-ft
```

The above command starts XCheck in running mode (option `-run xpath-ft`, where `xpath-ft` is the folder containing the experiment). XCheck runs each query once (option `-n 1`), saving the query answer (option `-s`), without generating any plot.

We say that an engine *supports* a query if the engine runs the query without errors. Moreover, an engine *correctly supports* a query if the engine supports the query and produces the right answer. The *functional completeness measure* is defined the number of queries that are supported by the engine divided by the number of queries in the test. The *functional correctness measure* is the number of queries that are correctly supported by the engine divided by the number of queries that are supported by the engine (whenever the engine does not support any query we set correctness to 0). The following table summarizes our findings:

engine	**completeness**	**correctness**
XSLTproc	66/66	65/66
Xalan-Java	66/66	66/66
SaxonB	66/66	66/66
Qizx/open	66/66	62/66

The problematic queries are F7 for XSLTproc and A9, A10, P9, and P10 for Qizx/open. In particular Qizx/open misinterprets the semantics of `following` and `preceding` axes. See XPathMark website [3] to compare the correct answer and the engine one for the above queries.

2.2 Experimenting with XPath Performance Test

We now move to the evaluation of performance. In this case there are several possible experiments, including query-by-query performance, aggregated performance, data and query scalability. First of all, we have to decide what to measure. The *query evaluation task* can be divided into different sub-tasks including: document loading and parsing, query compilation, query processing, result serialization. Ideally, a query processor should give timing information about all these tasks. The *total query evaluation time* (TET) is the time taken by the engine to complete the whole evaluation task. For each of these tasks we can measure the *elapsed real time* or the *user CPU time*. The latter is more accurate since it is independent of the workload of the CPU. XCheck captures and analysis the partial evaluation times whenever they are given by the engine. Furthermore, XCheck always measures TET as user CPU time.

In the following we discuss some typical performance experiments that we have done and show how to run them with XCheck. We measured the time for the complete evaluation of the query (TET) as user CPU time expressed in seconds.

Query-by-query performance. This experiment is meant to measure the query evaluation time for each query of the benchmark on a fixed document.

This is performed by describing the experiment (which queries, which documents, which engines) using an XML document and then by starting XCheck in *running mode*. In this mode XCheck executes the described experiment on the given sequence of engines. You can perform this with the following command:

```
$ ./XCheck.pl -p -run xpath-ft
```

The option `-p` is used to generate different types of plots depicting the query evaluation times for the different queries on the fixed document. By default, each query is executed 4 times. The first invocation is discarded and the time is computed as the average of the last 3 calls. You can change the number of query executions with option `-n number`, where `number` is the number of executions on which the average is computed, that is, `number` is 3 by default. You might also want to run this experiment on a subset of the benchmark or on a subset of the engine list (maybe because some queries are not supported by some engine). For instance, we ran this experiment on a 2 MegaByte XML document for all queries except those of XPath-F, which cannot be evaluated by XSLT processors Xalan and XSLTproc (Figure 4 at the end of the paper). Then, we ran the experiment on the same document for queries in XPath-F and engines SaxonB and Qizx/open (Figure 5 at the end of the paper). We ran XCheck in *data analysis mode*[1]. In this mode XCheck elaborates and/or plots the timing information computed during the running phase. To do so you have to configure the data analysis XML document (which subsets of queries and engines) and run XCheck as follows:

```
$ ./XCheck.pl -p -data xpath-ft
```

Aggregated query performance. You can exploit the data analysis mode also to generate some *aggregating measures*. For instance, you might want to compute the *averaged total evaluation time* (ATET) over all queries but XPath-F ones (Figure 6 at the end of the paper). Or you might be interested in computing *stability measure*, that is the ratio of the standard deviation and the mean on all queries but XPath-F (Figure 7 at the end of the paper). Showing a stable performance is good for an engine since the response time is predictable. You can run these experiments by properly configuring the data analysis XML document (which aggregating measures on which experiment components) an run XCheck in data analysis mode as follows:

```
$ ./XCheck.pl -p -data xpath-ft
```

Data scalability. This experiment is meant to evaluate the performance of the evaluation of a fixed query over a sequence of XML documents of increasing size. For instance, we fixed query E1 of XPath-PT and increase the

[1] Another possibility is to change the experiment and run XCheck with `update` option. However, in this way, you are going to loose the timing informations for the components that you have removed from the experiment. Use `update` option to run an experiment incrementally, not decrementally.

document size from 2 to 22 MegaBytes (Figure 8 at the end of the paper). To run this experiment with XCheck you have to configure an experiment XML document with the query and the document sequence. Notice that XCheck allows you generate the documents on-the-fly (assuming you have at disposal a document generator. For XPath-PT you can use XMark document generator). Then you can start XCheck in running mode as follows (`xpath-ds` is the folder of this experiment):

```
$ ./XCheck.pl -p -run xpath-ds
```

Query scalability.
This experiment is meant to evaluate the performance of the evaluation of a sequence of queries of increasing complexity over a fixed XML document. For instance, we fixed a document of 2 MegaBytes and tested the query sequence B13(i) of of XPath-PT (this is a parametric query), for $i = 0, 1, \ldots, 5$ (Figure 9 at the end of the paper). To run this experiment with XCheck you have to configure an experiment XML document with the query sequence and the document. Notice that XCheck allows you generate the queries on-the-fly (assuming you have at disposal a query generator. For each parametric query in XPath-PT we provide a Perl generator). Then you can start XCheck in running mode as follows (`xpath-qs` is the folder of this experiment):

```
$ ./XCheck.pl -p -run xpath-qs
```

2.3 Interpretation of the results

The crucial final step in the experimental experience consists in interpreting the results of the experiments. The outcomes of the functional test are quite straightforward to understand. For instance, it is clear from our tests that Qizx/open misinterprets `following` and `preceding` axes. It particular it considers descendant nodes as part of following ones and ancestor nodes as part of preceding ones.

Much more subtle is the interpretation of the performance experience. The difficulty is due to the abundance of possible experiments and to the fact that not all of them are meaningful. We refer to [10] for an interesting discussion of issues that arise when one attempts to analyze algorithms experimentally. As said above, our experimental analysis is meant to underline some typical experiments suggested by the proposed banchmark and is far from being exhaustive. This does not allow us to draw *general* conclusions about the performance behavior of the involved engines. Nevertheless, it allows us to notice the following:

– on relatively small documents (2 MegaBytes), XSLTproc is generally the fastest but have some bottlenecks for queries like B4, B9, B19, E5 involving `following` and `preceding` axes (see Figure 4);
– however, on average, SaxonB is the fastest, followed by XSLTproc, Xalan, and Qizx/open in this order (see Figure 6);

– SaxonB and Xalan show a very stable behavior. On the contrary, Qizx/open and in particular XSLTproc are not stable (see Figure 7);
– all the processors show linear data scalability on query E1. Moreover, on this query, XSLTproc is still the best performing also on bigger documents (see Figure 8);
– apparently, XSLTproc is the only engine that implements early node duplicate elimination. Late duplicate elimination might results in exponential blow-up as noticed in the pioneering paper [11] (see Figure 9).

References

1. Franceschet, M.: XPathMark: an XPath benchmark for XMark generated data. In: XSym. Volume 3671 of LNCS. (2005) 129–143
2. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A benchmark for XML data management. In: VLDB. (2002) 974–985 `http://www.xml-benchmark.org`.
3. Franceschet, M.: XPathMark. Functional and performance tests for XPath. `http://www.dimi.uniud.it/~francesc/xpathmark` (2006)
4. Benedikt, M., Koch, C.: XPath leashed. Submitted for publication (see authors' web pages for a draft version). (2006)
5. ten Cate, B.: The expressivity of XPath with transitive closure. In: PODS. (2006)
6. Marx, M.: Conditional XPath, the first order complete XPath dialect. In: PODS. (2004) 13–22
7. Authors: Extended XSLT. `http://www.exslt.org` (2006)
8. World Wide Web Consortium: XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath` (1999)
9. Afanasiev, L., Franceschet, M., Marx, M., Zimuel, E.: XCheck: A platform for benchmarking XQuery engines (demonstration). In: VLDB. (2006) `http://ilps.science.uva.nl/Resources/XCheck`.
10. Johnson, D.S.: A theoretician's guide to the experimental analysis of algorithms. In: Proceedings of DIMACS Implementation Challenges. (2002) 215–250
11. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. In: VLDB. (2002) 95–106

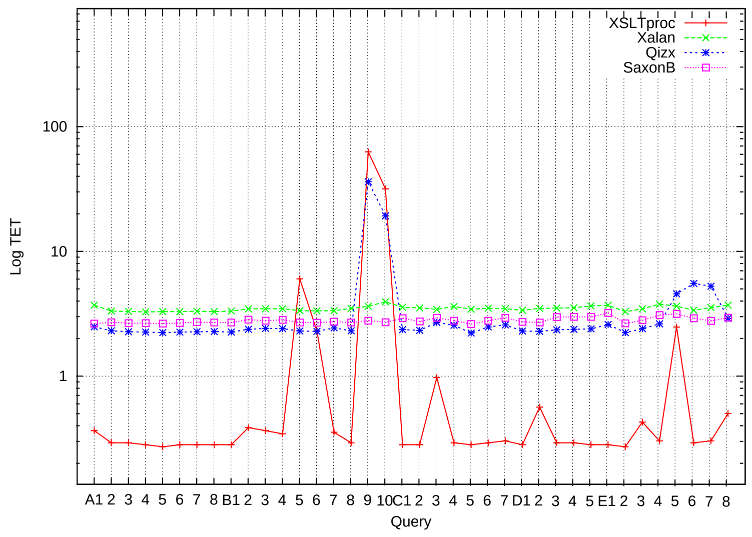

Fig. 4. Query performance (excluding XPath-F)

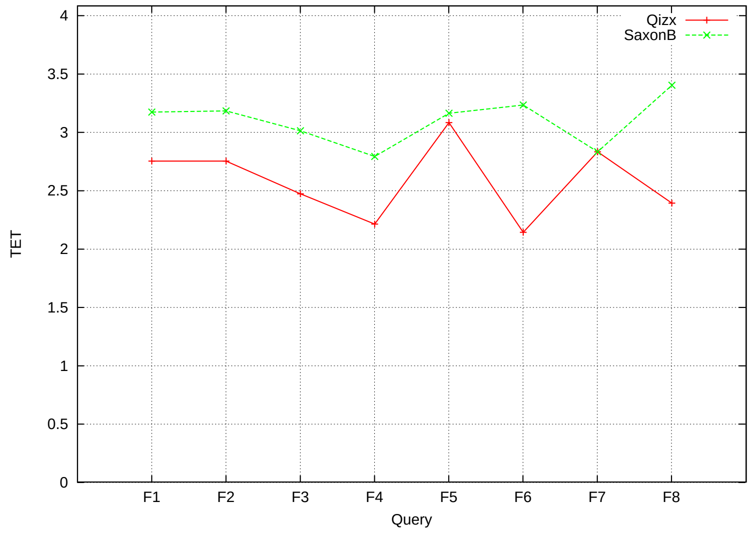

Fig. 5. Query performance on XPath-F

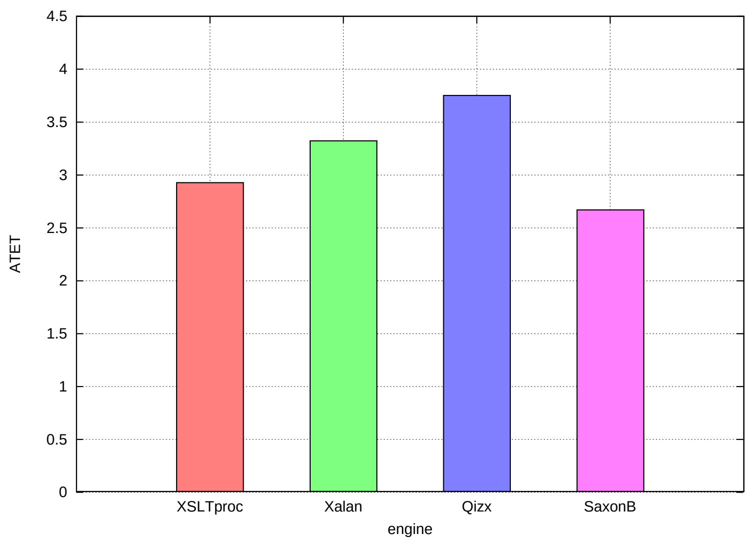

Fig. 6. Averaged query performance

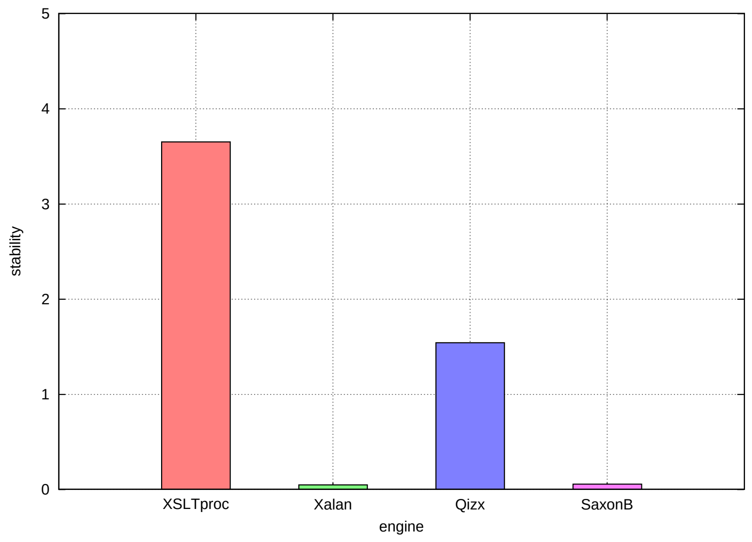

Fig. 7. Stability

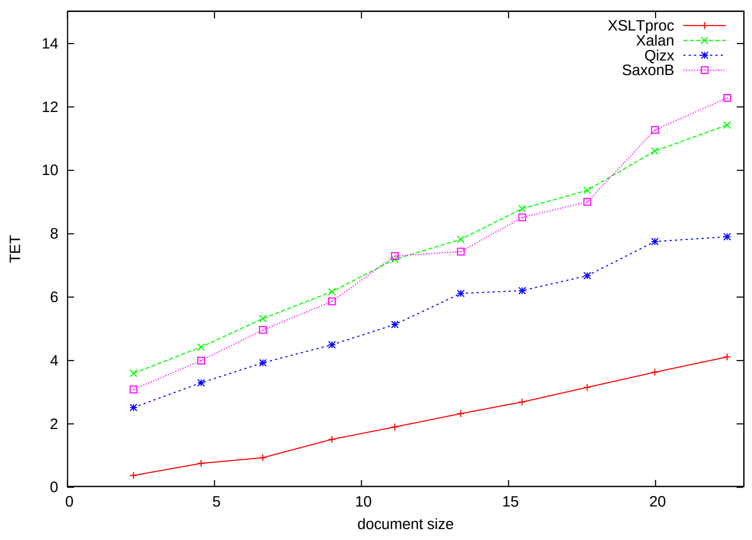

Fig. 8. Data scalability

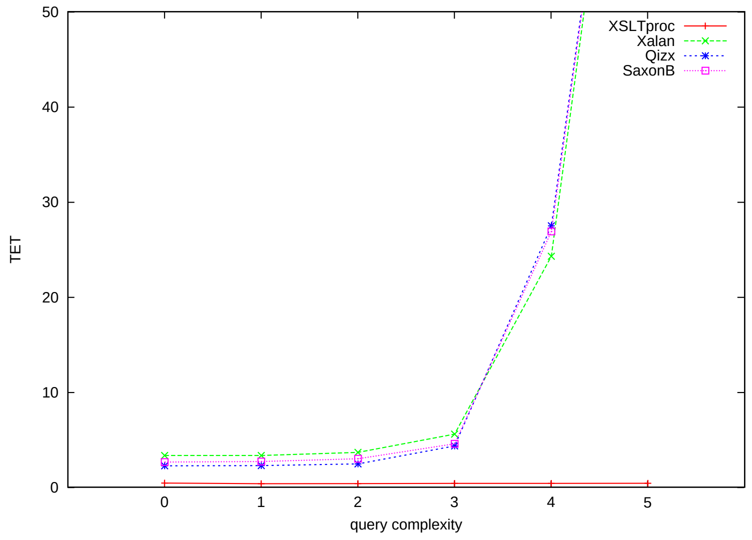

Fig. 9. Query scalability