XPathMark: An XPath Benchmark for the XMark Generated Data

Massimo Franceschet[1,2]

[1] Informatics Institute, University of Amsterdam
Kruislaan 403 – 1098 SJ Amsterdam, The Netherlands
[2] Dipartimento di Scienze, Università "Gabriele D'Annunzio"
Viale Pindaro, 42 – 65127 Pescara, Italy

Abstract. We propose XPathMark, an XPath benchmark on top of the XMark generated data. It consists of a set of queries which covers the main aspects of the language XPath 1.0. These queries have been designed for XML documents generated under XMark, a popular benchmark for XML data management. We suggest a methodology to evaluate the XPathMark on a given XML engine and, by way of example, we evaluate two popular XML engines using the proposed benchmark.

1 Introduction

XMark [1] is a well-known benchmark for XML data management. It consists of a scalable document database modelling an Internet auction website and a concise and comprehensive set of XQuery queries which covers the major aspects of XML query processing.

XQuery [2] is much larger than XPath [3], and the list of queries provided in the XMark benchmark mostly focuses on XQuery features (joins, construction of complex results, grouping) and provides little insight about XPath characteristics. In particular, only `child` and `descendant` XPath axes are exploited. In this paper, we propose XPathMark [4], an XPath 1.0 benchmark for the XMark document database. We have developed a set of XPath queries which covers the major aspects of the XPath language including different axes, node tests, Boolean operators, references, and functions. The queries are concise, easy to read and to understand. They have a natural interpretation with respect to the semantics of XMark generated XML documents. Moreover, we have thought most of the queries in such a way that the sizes of the intermediate and final results they compute, and hence the response times as well, increase as the size of the document grows. XMark comes with an XML generator that produces XML documents according to a numeric scaling factor proportional to the document size.

The targets of XPathMark are:

– *functional completeness*, that is, the ability to support the features offered by XPath;
– *correctness*, that is, the ability to correctly implement the features offered by XPath;

– *efficiency*, that is, the ability to efficiently process XPath queries;
– *data scalability*, that is, the ability to efficiently process XPath queries on documents of increasing sizes.

Since XPath is the core retrieval language for XSLT [5], XPointer [6] and XQuery [2], we think that the proposed benchmark can help vendors, developers, and users to evaluate these targets on XML engines implementing these technologies.

Our contribution is as follows. In Section 2 we describe the proposed XPath benchmark. In Section 3, we suggest how to evaluate the XPath benchmark on a given XML engine and, by way of example, we evaluate, using XPathMark, two popular XML engines, namely Saxon [7] and Galax [8]. Finally, in Section 4, we outline future work.

2 XPathMark: An XPath Benchmark for the XMark Generated Data

XPathMark has been designed in XML and is available at the XPathMark website [4]. In this section we describe a selection of the benchmark queries.

We first motivate our choice of developing the benchmark as XML data. This solution has all the advantages of XML [9]. In particular:

– the benchmark can be easily read, shipped, and modified;
– the benchmark can be queried with any XML query language;
– it is easier to write a *benchmark checker*, that is an application that automatically checks the benchmark against a given XML engine, that computes performance indexes, and that shapes the performance outcomes in different formats (plain text, XML, HTML, Gnuplot).

Figure 1 contains the Document Type Definition (DTD) for the XML document containing the benchmark. The root element is named `benchmark` and has the attributes `targets` (the targets of the benchmark, for instance, functional completeness), `language` (the language for which the benchmark has been written, for instance XPath 1.0), and `authors` (the authors of the benchmark). The benchmark element is composed of a sequence of `document` elements followed by a sequence of `query` elements. Each `document` element is identified by an attribute called `id` of type ID and contains, enclosed into a Character Data (CDATA) section, a possible target XML document for the benchmark queries. Each `query` element is identified by an attribute called `id` of type ID and has an attribute called `against` of type IDREF that refers to the document against which the query must be evaluated. Moreover, each `query` element contains the following child elements:

– `type`, containing the category of the query;
– `description`, containing a description of the query in English;
– `syntax`, containing the query formula in the benchmark language syntax;

```
<!ELEMENT benchmark     (document*,query*)>
<!ELEMENT document      (#PCDATA)>
<!ELEMENT query         (type,description,syntax,answer)>
<!ELEMENT type          (#PCDATA)>
<!ELEMENT description   (#PCDATA)>
<!ELEMENT syntax        (#PCDATA)>
<!ELEMENT answer        (#PCDATA)>

<!ATTLIST benchmark targets   CDATA #REQUIRED
                    language  CDATA #REQUIRED
                    authors   CDATA #REQUIRED>
<!ATTLIST document  id        ID #REQUIRED>
<!ATTLIST query     id        ID #REQUIRED
                    against   IDREF #REQUIRED>
```

Fig. 1. The benchmark DTD

– `answer`, containing the result of the evaluation of the query against the pointed document, enclosed within a CDATA section. The result is always a sequence of XML elements with no separator between two consecutive elements (not even a whitespace).

We have included in the benchmark two target documents. The first document corresponds to the XMark document generated with a scaling factor of 0.0005. A document type definition is included in this document. The set of queries that have been evaluated on this document are divided into the following 5 categories: axes, node tests, Boolean operators, references, and functions. In the following, for each category, we give a selection of the corresponding benchmark queries (see [4] for the whole query set). See [1] for the XMark DTD.

Axes. These queries focus on the navigational features of XPath, that is on the different kinds of axes that may be exploited to browse the XML document tree. In particular, we have the following sub-categories.

Child Axis. One short query (Q1) with a possibly large answer set, and a deeper one (Q2) with a smaller result. Only the child axis is exploited in both the queries.

Q1 *All the items*

```
/site/regions/*/item
```

Q2 *The keywords in annotations of closed auctions*

```
/site/closed_auctions/closed_auction/annotation/
description/parlist/listitem/text/keyword
```

Descendant Axes. The tag `keyword` may be arbitrarily nested in the document tree and hence the following queries can not be rewritten in terms of `child` axis. Notice that `listitem` elements may be nested in the document. During the processing of query Q4 an XPath processor should avoid to search the same subtree twice.

Q3 *All the keywords*

```
//keyword
```

Q4 *The keywords in a paragraph item*

```
/descendant-or-self::listitem/descendant-or-self::keyword
```

Parent Axis. Elements named `item` are children of the world region they belong to. Since XPath does not allow disjunction at axis step level, one way to retrieve all the items belonging to either North or South America is to combine the parent axis with disjunction at filter level (another solution is query Q22 that uses disjunction at the query level).

Q5 *The (either North or South) American items*

```
/site/regions/*/item[parent::namerica or parent::samerica]
```

Ancestor Axes. Elements named `keyword` may be arbitrarily deep in the document tree hence the ancestor operator in the following queries may have to ascend the tree of an arbitrarily number of levels.

Q6 *The paragraph items containing a keyword*

```
//keyword/ancestor::listitem
```

Q7 *The mails containing a keyword*

```
//keyword/ancestor-or-self::mail
```

Sibling Axes. Children named `bidder` of a given open auction are siblings, and the XPath sibling axes may be exploited to explore them. As for query Q4 above, during the processing of query Q9, the XPath processor should take care to visit each bidder only once.

Q8 *The open auctions in which a certain person issued a bid before another person*

```
/site/open_auctions/open_auction[bidder[personref/@person=
'person0']/following-sibling::bidder[personref/@person='person1']]
```

Q9 *The past bidders of a given open auction*

```
/site/open_auctions/open_auction[@id='open_auction0']
/bidder/preceding-sibling::bidder
```

Following and Preceding Axes. `following` and `preceding` are powerful axes since they may potentially traverse all the document in document or reverse document order. In particular, `following` and `preceding` generally explore more than `following-sibling` and `preceding-sibling`. Compare query Q8 with query Q11: while in Q8 only sibling bidders are searched, in Q11 also bidders of different auctions are accessed.

Q10 *The items that follow, in document order, a given item*

```
/site/regions/*/item[@id='item0']/following::item
```

Q11 *The bids issued by a certain person that precedes, in document order, the last bid in document order of another person*

```
/site/open_auctions/open_auction/bidder[personref/
@person='person1']/preceding::bidder[personref/@person='person0']
```

Node Tests. The queries in this category focus on node tests, which are ways to filter the result of a query according to the node type of the resulting nodes.

Q18 *The children nodes of the root that are comments*

```
/comment()
```

Q21 *The text nodes that are contained in the keywords of the description element of a given item*

```
/site/regions/*/item[@id='item0']/description//keyword/text()
```

Boolean operators. Queries may be disjuncted with the | operator, while filters may be arbitrarily combined with conjunction, disjunction, and negation. This calls for the implementation of intersection, union, and set difference on context sets. These operations might be expensive if the XPath engine does not maintain the context sets (document) sorted.

Q22 *The (either North or South) American items*

```
/site/regions/namerica/item | /site/regions/samerica/item
```

Q23 *People having an address and either a phone or a homepage*

```
/site/people/person[address and (phone or homepage)]
```

Q24 *People having no homepage*

```
/site/people/person[not(homepage)]
```

References. References turns the data model of XML documents from trees into graphs. A reference may potentially point to any node in the document having an attribute of type ID. Chasing references implies the ability of coping with arbitrary jumps in the document tree. References are crucial to avoid redundancy in the XML database and to implement joins in the query language. In summary, references provide data and query flexibility and they pose new challenges to the query processors.

Reference chasing is implemented in XPath with the function `id()` and may be static, like in query Q25, or dynamic, like in queries Q26-Q29. The `id()` function may be nested (like in query Q27) and its result may be filtered (like in query Q28). The `id()` function may also be used inside filters (like in query Q29).

Q25 *The name of a given person*

```
id('person0')/name
```

Q26 *The open auctions that a given person is watching*

```
id(/site/people/person[@id='person1']/watches/watch/@open_auction)
```

Q27 *The sellers of the open auctions that a given person is watching*

```
id(id(/site/people/person[@id='person1']
/watches/watch/@open_auction)/seller/@person)
```

Q28 *The American items bought by a given person*

```
id(/site/closed_auctions/closed_auction[buyer/@person='person4']
/itemref/@item)[parent::namerica or parent::samerica]
```

Q29 *The items sold by Alassane Hogan*

```
id(/site/closed_auctions/closed_auction
[id(seller/@person)/name='Alassane Hogan']/itemref/@item)
```

Functions. XPath defines many built-in functions for use in XPath expressions. The following queries focus on some of those.

Q30 *The initial and last bidder of all open auctions*

```
/site/open_auctions/open_auction
/bidder[position()=1 and position()=last()]
```

Q31 *The open auctions having more than 5 bidders*

```
/site/open_auctions/open_auction[count(bidder)>5]
```

Q36 *The items whose description contains the word 'gold'*

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

Q39 *Mails sent in September*

```
/site/regions/*/item/mailbox/mail
[substring-before(substring-after(date,'/'),'/')='09']
```

Q44 *Open auctions with a total increase greater or equal to 70*

```
/site/open_auctions/open_auction[floor(sum(bidder/increase))>=70]
```

XMark documents do not contain any comment or processing instruction. Moreover, they do no declare namespaces and language attributes. Although we have used these features in the first part of the benchmark, the corresponding queries do not give interesting insights when evaluated on XMark documents, since their answer sets are trivial. Therefore, we included in the benchmark a second document and a different set of queries in order to test these features only. For space reasons, we do not describe this part of the benchmark here and we invite the interested reader to consult the XPathMark website [4].

3 Evaluation of XML Engines

In this section we suggest how to evaluate XPathMark on a given XML engine. Moreover, by way of example, we evaluate, using XPathMark, two popular XML engines, namely Saxon [7] and Galax [8].

3.1 Evaluation Methodology

XPathMark can be checked on a set of XML processors and conclusions about the performances of the processors can be drawn. In this section, we suggest a method to do this evaluation.

We describe a set of *performance indexes* that might help the evaluation and the comparison of different XML processors that have been checked with XPathMark. We say that a query is *supported* by an engine if the engine processes the query without giving an error. A supported query is *correct* with respect to an engine if it returns the correct answer for the query. We define the *completeness index* as the number of supported queries divided by the number of benchmark queries, and the *correctness index* as the number of supported and correct queries divided by the number of supported queries. The completeness index gives an indication of how much of the benchmark language (XPath in our case) is supported by the engine, while the correctness index reveals the portion of the benchmark language that is correctly implemented by the engine.

XMark offers a document generator that generates XML documents of different sizes according to a numeric scaling factor. The document size grows linearly with respect to the scaling factor. For instance, factor 0.01 corresponds to a document of (about) 1,16 MB and factor 0.1 corresponds to a document of (about) 11,6 MB. Given an XMark document and a benchmark query, we can measure the time that the XML engine takes to evaluate the queries on the document. The *query response time* is the time taken by the engine to give the answer for the query on the document, including parsing of the document, parsing, optimization, and processing of the query, and serialization of the results. It might be interesting to evaluate the *query processing time* as well, which is the fraction of the query response time that the engine takes to process the query only, excluding the parsing of the document and the serialization of the results. We define the *query response speed* as the size of the document divided by the query response time. The measure unit is, for instance, MB/sec.

We may run the query against a documents series of documents of increasing sizes. In this case, we have a *speed sequence* for the query. The *average query response speed* is the average of the query response speeds over the document series. Moving from one document (size) to another, the engine may show either a positive or a negative *acceleration* in its response speed, or the speed may remain constant.

The concept of speed acceleration is intimately connected to that of *data scalability*. Consider two documents d_1 of size s_1 and d_2 of size s_2 in the document series with $s_1 < s_2$, and a query q. Let t_1 and t_2 be the response times for query q on documents d_1 and d_2, respectively. Let $v_1 = s_1/t_1$ be the speed of q over d_1 and $v_2 = s_2/t_2$ be the speed of q over d_2. The *data scalability factor* for query q is defined as:

$$\frac{v_1}{v_2} = \frac{t_2 \cdot s_1}{t_1 \cdot s_2}$$

If the scalability factor is lower than 1, that is $v_1 < v_2$, then we have a *positive speed acceleration* when moving from document d_1 to document d_2. In this case, we say that the scalability is *sub-linear*. If the scalability factor is higher than 1, that is $v_1 > v_2$, then we have a *negative speed acceleration* when moving from document d_1 to document d_2. In this case, we say that the scalability is *super-linear*. Finally, if the scalability factor is equal to 1, that is $v_1 = v_2$, then the speed is constant when moving from document d_1 to document d_2. In this case, we say that the scalability is *linear*. A sub-linear scalability means that the response time grows less than linearly, while a super-linear scalability means that the response time grows more than linearly. A linear scalability indicates that the response time grows linearly. For instance, if $s_2 = 2 \cdot s_1$ and $t_2 = 4 \cdot t_1$, then the scalability factor is 2 and the time grows quadratically on the considered segment.

Once again, we may run query q against series of documents of increasing sizes and generate a *data scalability sequence* for query q. The *average data scalability factor* for query q is the average of the data scalability factors for query q over the document series.

All these indexes can be computed for a single query or for an arbitrary subset of the benchmark. Of particular interest is the case when the whole benchmark is considered. Given a document d, we define the *average benchmark response time* for d as the average of the response times of all the benchmark queries on document d. Moreover, the *benchmark response speed* for d is defined as the size of d divided by the average benchmark response time. Notice that the benchmark response speed is different from the average of the response speeds for all the benchmark queries. Finally, the *data scalability factor for the benchmark* is defined as above in terms of the benchmark response speed. If we take the average of the benchmark response speed (respectively, data scalability factor for the benchmark) over a document series we get the *average benchmark response speed* (respectively, *average data scalability factor for the benchmark*). The former indicates how fast the engine processes XPath, while the latter reveals how well the engine scales-up with respect to XPath when the document size increases.

The outcomes of the evaluation for a specific XML engine should be formatted in XML. In Figure 2 we suggest a DTD for this purpose. The document root is named `benchmark`. The engine under evaluation and its version are specified as attributes of the element `benchmark`. The benchmark element has an `index` child and zero or more `query` children.

The `index` element contains the performance indexes of the engine and has attributes describing the testing environment. The testing environment contains

```
<!ELEMENT benchmark        (indexes,query*)>

<!ELEMENT indexes          (completeness,correctness,times?,speeds?,
                            scalas?,avgspeed?,avgscala?)>
<!ELEMENT completeness     (#PCDATA)>
<!ELEMENT correctness      (#PCDATA)>
<!ELEMENT times            (time+)>
<!ELEMENT speeds           (speed+)>
<!ELEMENT scalas           (scala+)>
<!ELEMENT time             (#PCDATA)>
<!ELEMENT speed            (#PCDATA)>
<!ELEMENT scala            (#PCDATA)>
<!ELEMENT avgspeed         (#PCDATA)>
<!ELEMENT avgscala         (#PCDATA)>

<!ELEMENT query            (type,description,syntax,supported,error?,
                            correct,given_answer?,expected_answer?,
                            times?,speeds?,scalas?,avgspeed?,avgscala?)>
<!ELEMENT type             (#PCDATA)>
<!ELEMENT description      (#PCDATA)>
<!ELEMENT syntax           (#PCDATA)>
<!ELEMENT supported        EMPTY>
<!ELEMENT error            (#PCDATA)>
<!ELEMENT correct          EMPTY>
<!ELEMENT given_answer     (#PCDATA)>
<!ELEMENT expected_answer  (#PCDATA)>

<!ATTLIST benchmark engine    CDATA #REQUIRED
                    version   CDATA #REQUIRED>
<!ATTLIST query     id        ID #REQUIRED>
<!ATTLIST indexes   cpu       CDATA #IMPLIED
                    memory    CDATA #IMPLIED
                    os        CDATA #IMPLIED
                    time_unit (msec | csec | dsec | sec)  #IMPLIED
                    time_type (response | processing) #IMPLIED>
<!ATTLIST supported value     (yes | no) #REQUIRED>
<!ATTLIST correct   value (yes | no | undef) #REQUIRED>
<!ATTLIST time      factor    CDATA #REQUIRED>
<!ATTLIST speed     factor    CDATA #REQUIRED>
<!ATTLIST scala     factor1   CDATA #REQUIRED
                    factor2   CDATA #REQUIRED>
```

Fig. 2. The DTD for a benchmark outcome

information about the processor (`cpu`), the main memory (`memory`), the operating system (`os`), the time unit (`time_unit`), and the time type, that is, either response or processing time (`time_type`). The performance indexes are: the completeness index (`completeness`), the correctness index (`correctness`), a sequence of average benchmark response times for a document series (`times`, a sequence of `time` elements), a sequence of benchmark response speeds for a document series (`speeds`, a sequence of `speed` elements), a sequence of data scalability factors for the benchmark for a document series (`scalas`, a sequence of `scala` elements), the average benchmark response speed (`avgspeed`), and the average data scalability factor for the benchmark (`avgscala`). Each element of type `time` and `speed` has an attribute called `factor` indicating the factor of the XMark document on which it has been computed. Moreover, each element of type `scala` has two attributes called `factor1` and `factor2` indicating the factors of the two XMark documents on which it has been computed.

Each `query` element contains information about the single query and is identified by an attribute called `id` of type ID. In particular, it includes the category of the query (`type`), a description in English (`description`), the XPath syntax (`syntax`), whether or not the query is supported by the benchmarked engine (`supported`, it must be either `yes` or `no`), the possible error message (`error`, only if the query is not supported), whether or not the query is correctly implemented by the benchmarked engine (`correct`, it must be either `yes`, `no`, or `undef`. The latter is used whenever the query is not supported), the given and expected query answers (`given_answer` and `expected_answer`. They are used for comparison only if the query is not correct), a sequence of query response times for a document series (`times`, as above), a sequence of query response speeds for a document series (`speeds`, as above), a sequence of data scalability factors for the query for a document series (`scalas`, as above), the average query response speed (`avgspeed`), and the average data scalability factor for the query (`avgscala`).

The solution of composing the results in XML format has a number of advantages. First, the outcomes are easier to extend with different evaluation parameters. More importantly, the outcomes can be queried to extract relevant information and to compute performance indexes. For instance, the following XPath query retrieves the benchmark queries that are supported but not correctly implemented:

```
/benchmark/query[supported="yes" and correct="no"]/syntax
```

Moreover, the following XQuery computes the completeness and correctness indexes:

```
let $x := doc("outcome_engine.xml")/benchmark/query
let $y := $x[supported="yes"]
let $z := $x[correct="yes"]
return <indexes>
         <completeness> {count($y) div count($x)} </completeness>
         <correctness> {count($z) div count($y)} </correctness>
       </indexes>
```

Finally, the following XQuery computes the average query response time of queries over the axes category when evaluated on the XMark document with scaling factor 0.1:

```
let $x := doc("outcome_engine.xml")/
          benchmark/query[type="axes" and correct="yes"]
let $y := sum($x/times/time[@factor="0.1"])
let $z := count($x)
return <average_time> {$y div $z} </average_time>
```

More generally, one can easily program a benchmark checker that automatically tests and evaluates different XML engines with respect to XPathMark.

3.2 Evaluating Saxon and Galax

We ran the XPathMark benchmark on two state-of-the-art XML engines, namely Saxon [7] and Galax [8]. Saxon technology is available in two versions: the basic edition Saxon-B, available as an open-source product, and the schema-aware edition Saxon-SA available on a commercial license. We tested Saxon-B 8.4, with Java 2 Platform, Standard Edition 5.0. Galax is the most popular native XQuery engine available in open-source and it is considered a reference system in the database community for its completeness and adherence to the standards. We tested version 0.5. We ran all the tests on a 3.20 GHz Intel Pentium 4 with 2GB of main memory under Linux version 2.6.9-1.667 (Red Hat 3.4.2-6.fc3). All the times are response CPU times in seconds. For each engine, we ran all the supported queries on XMark documents of increasing sizes. The document series is the following (XMark factors):

(0.001, 0.002, 0.004, 0.008, 0.016, 0.032, 0.064, 0.128, 0.256, 0.512, 1)

corresponding to the following sizes (in MB):

(0.116, 0.212, 0.468, 0.909, 1.891, 3.751, 7.303, 15.044, 29.887, 59.489, 116.517)

It is worth noticing that in the computation of the completeness index we did not consider queries using the `namespace` axis, since this axis is no more supported in XQuery [2].

The whole evaluation outcomes can be accessed from the XPathMark website [4]. This includes the outcomes in XML for both the engines and some plots illustrating the behaviour of the performance indexes we have defined in this paper. In order to compare efficiency and scalability of the two engines, we also evaluated the subset of the benchmark corresponding to the intersection of the query sets supported by the two engines (which are different). This common base is the query set {Q1-Q9,Q12,Q13,Q15-Q24,Q30-Q47} of cardinality 39. In the following we report about our findings.

1. **Completeness and Correctness.** The completeness and the correctness indexes for Saxon are both 1, meaning that Saxon supports all the queries in the benchmark (excluding queries using the `namespace` axis, which are not counted) and all supported queries give the correct answer. The completeness index for Galax is 0.85. In particular, the axes `following` and `preceding` (which are in fact optional in XQuery) and the `id()` function

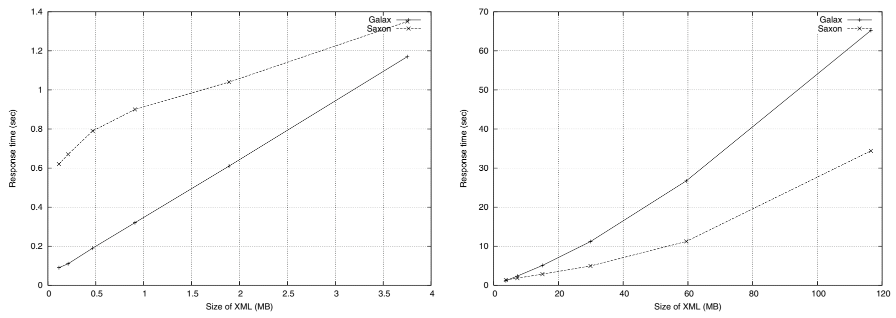

Fig. 3. Average benchmark response times

are not supported by Galax. However, all the supported queries give correct answers, hence the correctness index for Galax is 1.

2. **Efficiency.** On the common query set, the average benchmark response speed for Saxon is 2.80 MB/sec and that for Galax is 2.50 MB/sec. This indicates that Saxon is faster than Galax to process the (checked subset of the) benchmark. The average response time for a query in the benchmark, varying the document size, is depicted in Figure 3 (left side is from factor 0.001 to factor 0.032 and right side is from factor 0.032 to factor 1). Interestingly enough, Galax outperforms Saxon in the first track, corresponding to small documents (up to 3.7 MB), but Saxon catches up in the second track, corresponding to bigger documents. This trend is confirmed by the behaviour of the benchmark response speeds (see Figure 4 corresponding to the same segments of the document series).
3. **Scalability.** On the common query set, the average data scalability factor for the checked benchmark is 0.80 in the case of Saxon and it is 0.98 in the case of Galax. This indicates that Saxon scalas-up better than Galax as the size of the XML document increases. Figure 5 compares the data scalability factors for the two engines. Notice that Saxon's scalability is sub-linear up to XMark factor 0.256 (29.9 MB), and it is super-linear for bigger files. Galax's scalability is sub-linear up to XMark factor 0.032 (3.7 MB), and it is super-linear for bigger documents. This trend is confirmed by the behaviour of the benchmark response speeds (Figure 4). In particular, notice that Saxon's response speed increases (with a decreasing derivative) up to XMark factor 0.256, and then it decreases, while Galax has a positive acceleration up to XMark factor 0.032, and then the acceleration becomes negative. From this analysis, we conclude that, under our testing environment, Saxon is well performing up to a *break point* corresponding to an XML documents of size 29.9 MB, while the break point for Galax corresponds to a smaller file of 3.7 MB.

Finally, Figures 6 and 7 depict, for each query, the average response speeds and the average data scalability factors over the document series. Interestingly, the *qualitative* behaviour of the response speeds is the same for both the engines,

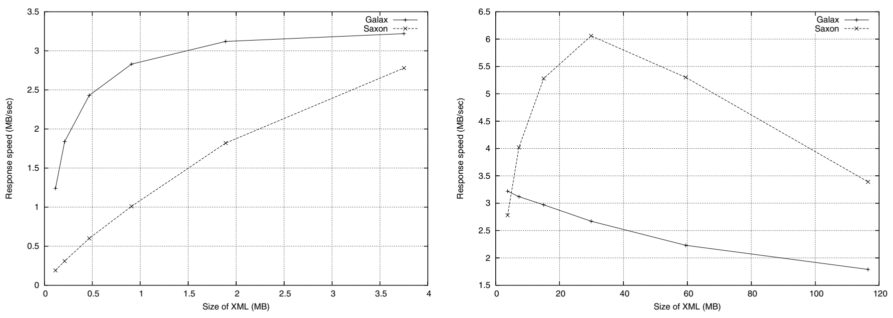

Fig. 4. Benchmark response speeds

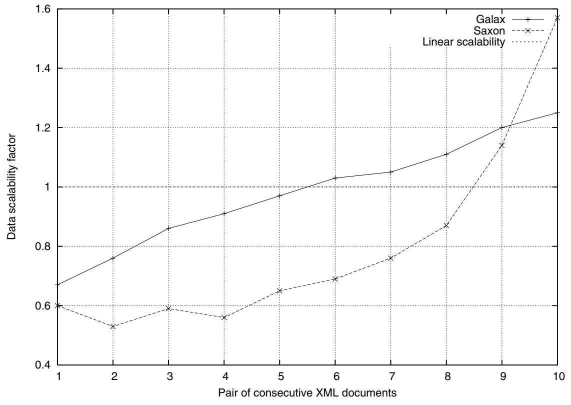

Fig. 5. Data scalability factors for the benchmark

with Saxon outperforming Galax in all the queries but Q35 (The elements written in Italian language: `//*[lang('it')]`). This might indicate that the two engines implement a similar algorithm to evaluate XPath queries. The data scalability factor for Galax is almost constant for all the queries, and it is less but close to linear scalability. The data scalability factor for Saxon is less stable. It is far below linear scalability for all the queries but the problematic Q35. In particular, the scalability factor for Q35 is higher than 3 in the last segment of the document series, indicating that the response time for Q35 grows more then quadratically (probably Saxon doesn't understand Italian very well!). Notice that Q35 is not problematic in Galax.

4 Future Work

We intend to improve XPathMark in different directions by: (i) enlarging the benchmark query set. In particular, we are developing a benchmark to test *query scalability*, that is the ability of an XML engine to process queries of increas-

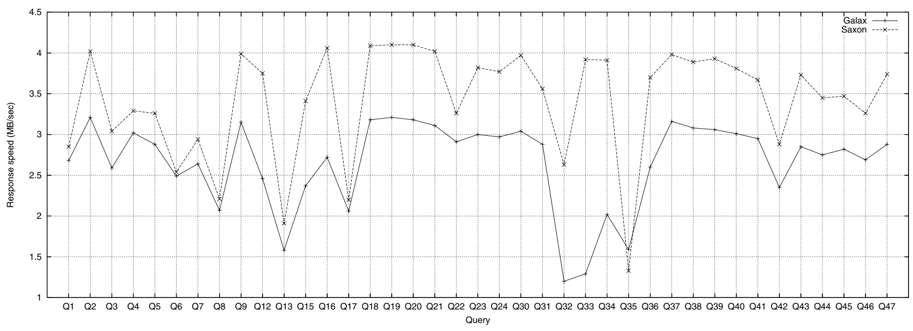

Fig. 6. Average query response speeds

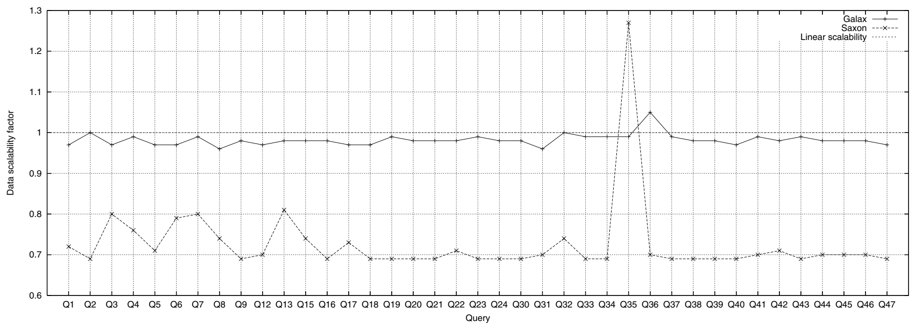

Fig. 7. Average data scalability factors for queries

ing lengths; (ii) studying different performance indexes to better evaluate and compare XML engines; (iii) implementing a *benchmark checker* in order to automatically compare the performance of different query processors with respect to XPathMark.

XPathMark can also be regarded as a benchmark for testing the navigational fragment of the XQuery language in isolation. Indeed, XQuery crucially uses XPath to navigate XML trees, saving the retrieved node sequences into variables that may be further elaborated by, e.g., joining, sorting, and filtering. In this respect, XPathMark can be considered as a fragment of a new version of the XMark benchmark or as a part of a bigger benchmark evaluation project for XQuery (e.g., the micro-benchmark repository for XQuery proposed in [10]).

References

1. Schmidt, A.R., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In: Proceedings of the International Conference on Very Large Data Bases (VLDB). (2002) 974–985 `http://monetdb.cwi.nl/xml/`.

2. World Wide Web Consortium: XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery` (2005)
3. World Wide Web Consortium: XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath` (1999)
4. M. Franceschet: XPathMark: An XPath benchmark for XMark. `http://www.science.uva.nl/~francesc/xpathmark` (2005)
5. World Wide Web Consortium: XSL Transformations (XSLT). `http://www.w3.org/TR/xslt` (1999)
6. World Wide Web Consortium: XML Pointer Language (XPointer). `http://www.w3.org/TR/xptr` (2002)
7. Kay, M.H.: Saxon. An XSLT and XQuery processor. `http://saxon.sourceforge.net` (2005)
8. Fernández, M., Siméon, J., Chen, C., Choi, B., Gapeyev, V., Marian, A., Michiels, P., Onose, N., Petkanics, D., Ré, C., Stark, M., Sur, G., Vyas, A., Wadler, P.: Galax. The XQuery implementation for discriminating hackers. `http://www.galaxquery.org` (2005)
9. Harold, E.R., Means, W.S.: XML in a Nutshell. 3rd edn. O'Reilly (2004)
10. Afanasiev, L., Manolescu, I., Michiels, P.: MemBeR: a micro-benchmark repository for XQuery project description. In: Proceedings of the International XML Database Symposium (XSym). (2005)