XPath Engine Performance under Equivalent Expressions

Höskuldur Hlynsson, Christian Kissig, and Laura Rimell

March 24, 2005

Contents

1 Introduction **1**
1.1 XPath . . . 2
1.2 Goals and Design of this Benchmark . . . 2

2 Theoretical Preliminaries **3**
2.1 Satisfiable XPath Terms . . . 3
2.2 Notions of Equivalence . . . 4

3 Planning the Evaluation **4**
3.1 Collections of Queries vs. Automatic Generation . . . 4
3.2 Query Sets . . . 5
3.2.1 Initial Query Set . . . 5
3.2.2 Generation by Rewriting . . . 6
3.3 Classes of Queries . . . 7
3.3.1 Automatically Generated Queries . . . 7
3.3.2 Manually Constructed DTD/Schema-Aware Queries . . . 11

4 Performing the Benchmark **12**
4.1 Environment . . . 12
4.2 Generation of Input . . . 12
4.3 Interpreting the Results . . . 12

5 Results **13**
5.1 Test set 1 . . . 13
5.2 Test set 2 . . . 14

6 Conclusion **15**

1 Introduction

XPath (XML Path Language, see [10]) is a member of the XML family of languages. (See also [2] and [7] for further information about XML and related languages.) As storage of data in XML, both on the internet and in offline databases, becomes more common, the need for efficient retrieval of structured XML data becomes greater. XPath is a key component in XML query languages

such as XQuery, as well as transformation/formatting languages such as XSLT. As such, it is important for XML query engines to process XPath expressions efficiently. XPath is also a language with a simple, yet rich, syntax, in which there are multiple possible ways to express a given query. This benchmark is designed to assist developers in ensuring that variants of an XPath query expression do not affect processing time. Although there are several existing benchmarks for XPath processing efficiency in general (see [9] and [8], for example), there is no existing benchmark, to our knowledge, which addresses this particular aspect of the XPath standard.

1.1 XPath

XML is a markup language, or metalanguage, in which tags identify data. An XML document is a tree-structured (hierarchical) collection of nodes. The popularity of XML stems from the fact that XML allows the flexible development of user-defined document types. It provides a robust, non-proprietary, persistent, and verifiable file format for the storage and transmission of text and data both on and off the Web; and it removes the more complex options of SGML, making it easier to program for (adopted from [3]).

XPath is a language used to address nodes in an XML document. The core part of XPath, also known as *core* or *navigational* XPath, uses a slash-separated list of element names that describes a path through the XML document. Additional parts of XPath allow addressing of element attributes, as well as predicates that refine the navigational searches. Predicates may address the structure of the XML document and also include basic numeric and string comparisons.

In this example, one can see the slash indicating the child relation and the square brackets indicating predicates (cf. indexing information which is often represented by square brackets).

- para/list[@type="ordered"][3] – selects all list elements of type "ordered" that are children of para elements, and returns the third list in each para.
- list[3][@type="ordered"] – selects the third LIST element, but only if it is of "ordered" type.

1.2 Goals and Design of this Benchmark

Within XPath, there are a number of different ways to express a given query. To begin with, XPath provides a set of *axes* which define relations between nodes in the XML document. These axes, including parent, child, ancestor, descendant, and sibling relations, allow for many redundancies. In addition, relations can be described with predicates as well as with paths. As a result, each query, in priciple, has countably many equivalent formulations. This benchmark is designed to bring out just such redundancies.

We have identified several general categories in which redundancies may occur. Within each category, we have attempted to define an exhaustive list of redundancy types. Each redundancy type is represented in the benchmark by a pair of equivalent queries. For each pair, we discuss why they may not have the same processing time.

Software that interprets XPath queries on XML documents is called an XPath query engine. If two query paths yield the same result, but one takes

longer to process, this can be a problem for an XPath processing engine. Developers of XPath query engines can use the benchmark to ensure that their engines do not process one member of a pair more slowly than the other. If this is the case, a small amount of preprocessing on the troublesome query should allow it to be processed more efficiently.

The queries are written against an XMark document (see [9]) which was generated using the XMark data generator. The XMark benchmarking suite for XML applications includes a DTD, the data generator, and a set of queries designed to test the capabilities of XML processing engines across a range of scenarios. We chose to base our queries on XMark documents because they are already in common use among XML benchmarking applications, so it will require minimal effort for many developers to add our benchmark to an existing test routine.

The XMark DTD describes an internet auction website. It includes elements for people, with their profiles, credit card information, etc.; open and closed auctions, with bids, bidders, and item descriptions; and categories of items, among other elements. For the full XMark DTD see [9].

The benchmark is also designed according to the criteria for a successful domain-specific benchmark as described by [4]: relevance, portability, scalability, and simplicity.

For simplicity, we do not address namespaces within the XPath standard.

2 Theoretical Preliminaries

XPath can be seen as a logic to reason over XML. In fact it is considered to be related closely to Modal Logics. So, it is not surprising that for the well-known constructions in XPath one can find model theoretic interpretations, like Signatures, Models, Terms, Formulae, Satisfiability, and Equivalence.

The signatures are determined by the XML specifciation, while the XML documents form the models of such a signature. DTDs or XSchemas impose constraints on those models, i.e. on those XML documents. XPath can then be seen as a logic allowing us to form terms over the XML signature. Those terms, or XPath queries, may be interpreted with respect to an XML document to address subtrees thereof. The interpretation is determined by the XPath Semantics.

In the following we introduce the common notions of satisfiability and equivalence along XPath queries.

2.1 Satisfiable XPath Terms

One must distinguish three categories of satisfiability for an XPath query. A query is satisfiable in general if it is interpreted to a non-empty subtree of any XML document. It is satisfiable w.r.t. a DTD/XSchema if such an XML document fulfills the constraint imposed by the DTD/Xschema. And, a query is satisfiable w.r.t. a specific XML document if it is interpreted to a non-empty subtree thereof.

2.2 Notions of Equivalence

Similar to the Satisfiability of XPath queries we can define a notion of Equivalence which actually splits up into three different flavours depending whether it is interpreted with regard to a DTD/XSchema, a specific XML document or none.

1. Total Equivalence: Two XPath terms are considered totally equivalent iff they are interpreted to the same set of elements in every model of the XML-specification
2. Equivalence w.r.t. a DTD/Schema: Two XPath terms are said to be equivalent w.r.t. a DTD/Schema iff they are interpreted to the same set of nodes for every XML-document satisfying the constraints given by the DTD/XSchema
3. Equivalence w.r.t. an XML-document: Two XPath terms are said to be equivalent w.r.t. an XML-document iff they are interpreted to the same set of nodes in the respective XML-document.

One can easily see that 1. $\Rightarrow$ 2. and 1. $\Rightarrow$ 3. If XML-documents are thought of as being implicitely associated with a DTD/XSchema one can also show that 2. $\Rightarrow$ 3.. Especially, the second implication is vital when showing equivalence of terms for some XML-document. Obviously, all unsatisfiable terms are equivalent. We call this special case trivial equivalence.

3 Planning the Evaluation

The goal of our evaluation is to exhibit differences between the XPath processing engines Galax 0.4.0 and Saxon 8.2 in how they exploit a priori knowledge about equivalences along XPath terms.

Both engines process the XPath queries in a two stage fashion. First, the query itself is parsed and translated into an internal representation. Galax makes use of the XML/XPath/XSLT-library yaxi for OCaML while Saxon's datastructures are derived from James Clark's xt library. However, both APIs have in common that queries in the rich syntax of XPath are reduced to a small sublanguage. Thereby the queries are normalized. After normalization the query is processed for a loaded document. Our benchmark should respect those two stages. It is common to Galax and Saxon that both apply naive, hence quick, rewriting to the XPath queries. The major part of complexity observed for query processing seems to be caused by the processing stage.

3.1 Collections of Queries vs. Automatic Generation

Any benchmark tests the runtime behaviour of a system for a certain input. Basically, there are two options to create the input for the evaluation. One may create an assorted collection of inputs representing what has to be tested for as closely as possible. Or, one may create a generator producing the input according to a certain scheme.

If the benchmark for an XPath engine is intended to test how fast the engine processes queries occuring often in practice, then one may simply collect queries

from real-life runs. However, one may miss important aspects if one is not aware of every application of the engine.

On the other hand an automatic generator of queries depends on a given scheme not to generate boring queries and thereby unnecessarily slowing down the benchmark. Such a scheme can either be created manually in some esoteric procedure, or semi-manually by comparing naturally occuring queries. Both attempts expose the creator or user of the benchmark to the danger of missing important aspects.

We decided to follow both paths. We will first give a suggestion on how to design such a generator for interesting XPath queries. But, the actual comparison of the XPath engines will be done both with queries constructed manually and by applying the suggested generation rules.

3.2 Query Sets

Our benchmark exposes runtime differences in processing equivalent XPath queries, such that the input generation aims at producing equivalent queries. Note that XPath is sub-language of first order logic (FOL) such that equivalent XPath queries are enumerable.

However, we decided against the generation of equivalent XPath queries from an FOL representation for several reasons. The FOL representation of XPath constructions has not been properly defined yet. The set of equivalent FOL formulae is uncomparably larger than the set of equivalent XPath queries, such that the generator would produce an incredible amount of FOL formulae being filtered out again. Thereby the generation process would be blown up artificially. Assume we have a set of equivalent XPath queries produced from their FOL representation. Then most of them could be filtered out because they do not match the goal of the benchmark. It is furthermore not possible to weight those generated queries for being interesting or for a certain categorization. The latter one allows the vendor of an XPath engine to improve their engine in a goal-oriented way.

Our suggestion for a generator is as follows. The generator starts with some initial query showing aspects that might be interesting to test for. To those initial queries the generator applies pre-defined rewrite rules in order to produce queries equivalent to the initial ones. Again the rewrite rules are chosen for representing test-worthy aspects.

3.2.1 Initial Query Set

As indicated before the initial queries are one of the most important components of the generation process. They do not only influence the evaluation result, but also the applicability of the rewrite rules. We proposed the following properties those initial queries should meet.

- *Naturalness*: An initial term should be natural in the sense that it could be written by a human, for example a webmaster, or generated by a transformation process implemented as an XSLT- or CGI-script.
- *Satisfiabilty*: Most of the terms should be satisfiable to avoid overevaluation of trivial equivalences.

- *Unsatisfiabilty*: There should also be some unsatisfiable terms. Their unsatisfiability should be decidable in the first half of the term to test whether the XPath engine under consideration interupts computation when crossing that point.
- *Size*: Initial terms should be of non-trivial size.
- *Diversity*: Initial terms should cover most constructions from the XPath 1.0 Spec. Furthermore, they should be tailored so that the rewrite rules will apply.

3.2.2 Generation by Rewriting

Starting from the initial terms we generate equivalent XPath terms in a term rewriting system (TRS) like manner, i.e. we apply certain rewrite rules at arbitrary positions in initial terms. These rewrite rules have to preserve various kinds of equivalence, depending on which notion of e Starting from the initial terms we generate equivalent XPath terms in a term rewriting system (TRS) like manner, i.e. we apply certain rewrite rules at arbitrary positions in initial terms. These rewrite rules have to preserve various kinds of equivalence, depending on which notion of equivalence is adopted.

The application of a rewrite rule is determined by the following parameters:

- the rewrite-rule itself, and thus its category as well
- the position of application: Rewriting a term at the root probably has more effect than rewriting at a deeper level, or even in an unsatisfiable subterm.
- the number of applications: One and the same rule may be applied at different positions in a expression. Some equivalences allow rewrites of a term at the same position, while others can build 'cycles' and thus allow alternating application of converse rewrite rules. In most cases a term is amenable to application of rules at different positions.

It is worth a mention that the above issues can be expressed in some kind of metric [1]. Such a metric is determined by some measure on the XPath query, the XML document, some weighting of the rewrite rules towards the intention of our benchmark, position and number of applications of rewrite rules. It is obviously hard to construct equivalent queries equally distant in terms of such a metric, as it is hard to define such a metric matching the intentions of a benchmark.

The following we will introduce the classes of rewrite rules that we identified.

[1] Mathematically, a *metric* is a function $d : X \times X \longrightarrow \mathbf{R}$ such that for every $x, y, z \in X$,

- $d(x, y) \geq 0$, with equality if and only if $x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$

3.3 Classes of Queries

In choosing rewrite rules, we have attempted to cover a broad range of the possibilities available in XPath, although we do not make any claims that the rewrite rules are fully exhaustive.

Each XPath query returns a node set. The possible criteria for determining whether a given node is a member of the target set are either structural, arrived at by navigating through the XML document tree, and/or predicate-based, testing whether some predicate holds at the node. Such predicates may again involve structural relations, or they may involve tests of the string or numeric content of the node. They may also involve the position of the node in relation to its siblings, in which case counting functions are used. These types of critera cover all the possibilities for queries in XPath. Thus, we have been able to divide our queries into a few main classes.

The first class is the navigational queries. There are many possible path descriptions leading to the same node set: for example, one might ask for the siblings of the current node, or for the children of the current node's parent. This type of redundancy is built into the XPath language to allow flexibility in writing queries, but it is clear that different ways of specifying a path may not lend themselves to processing with equal efficiency. The navigational class of equivalent queries is designed to test whether the processing engine can recognize simple equivalences between paths. In addition, in XPath it is often possible to express the same criterion using either the path syntax or the predicate syntax. We test whether these two types of queries are processed with the same efficiency. (This last type of query could as easily have been listed in the Predicates class, since it involves a comparison between navigation with path syntax and with predicate syntax.)

The second class of equivalent queries is predicates. Within a predicate, there may be multiple equivalent ways of expressing a test on a node. This class of queries is designed to test for equivalences within the predicate syntax.

The third class of queries involves boolean equivalences. Some queries, or parts of queries, will evaluate to a tautology or a contradiction. This class of queries tests whether the processing engine can recognize basic tautologies or contradictions within the query. Such equivalences are not unique to XML data but may occur in any kind of query language. Nevertheless, it is an important area to test since a processing engine will be much more efficient if it can short-circuit the performance of a long query by recognizing quickly that the query contains a tautology or contradiction.

We also identified two classes of queries that we did not include in the benchmark, but which the developer of an XPath query engine may wish to use as a sanity check. These are notational equivalences, including abbreviations and implicit vs. explicit type-casting; and counting equivalences, involving the counting functions available in the predicate syntax. We expect all such equivalences to be addressed by the normalization routines of any engine.

Manually constructed queries are addressed below.

3.3.1 Automatically Generated Queries

The rewrite rules make use of the following typed variables:

Variable	Type
RLP	RelativeLocationPath
LS,LS1,LS2	LocationStep
P,P1,P2	Predicate
E	Expr
A	Attribute
ALP	AbsoluteLocationPath
X	Axis
N,N1,N2	NameTest

The following are the classes of queries:

Navigational Equivalences based on navigational steps

1. Redundant use of antagonistic axes

   LS/N → LS/child::*/parent::*/N
   N/RLP → N/parent::*/child::*/RLP

   These rewrite rules take advantage of the opposition between the child and parent axes to create a path with needless back-and-forth. Similar rewrite rules can be defined for ancestor and descendant, preceding-sibling and following-sibling, etc. Note that the rules are defined to avoid problems at leaf and root nodes.

   Sample query pair:

   ```
   /site/people/child::*/watches/watch/@open_auction
   /site/people/child::*/child::*/parent::*/watches/watch/@open_auction
   ```

2. Redundant predicate

   N1/N2 → N1[child::N2]/N2

   This rewrite rule redundantly checks the presence of the N2 child using a predicate, before N2 is located in the path.

   Sample query pair:

   ```
   /site/people/child::*/watches/watch/@open_auction
   /site/people/child::*/watches[child::watch]/watch/@open_auction
   ```

3. Reverse vs. forward axes

   N2/N1/RLP → N1[parent:N2]/RLP
   N2[following::N1] → N1/preceding::N2

   These rewrite rules cause either a re-check of the parent element after the tree has already been traversed (in the case of top-down traversal), or a re-check of the preceding nodes. The inverse of these rules are important for streaming XML processing, where it is desirable to convert reverse axes into forward axes (see [5] for discussion of how this can be done efficiently).

   Sample query pair:

   ```
   //people/person/watches/watch/@open_auction
   //person[parent::people]/watches/watch/@open_auction
   ```

4. Paths instead of predicates

   N1[N2]/RLP → N1/N2/parent::N1/RLP

This rewrite rule changes a predicate into a path, causing the original node to be looked at twice.

Sample query pair:

```
//people[child::*]//watches/watch/@open_auction
//people/child::*/parent::people//watches/watch/@open_auction
```

5. Navigation among siblings

   parent::*/child::N/RLP → (preceding-sibling::N|self::N|following-sibling::N)/RLP

   This rewrite rule shows the equivalence between identifying siblings by their common parent, or by the sibling axes.

   Sample query pair:

```
/site/categories/category/name/parent::*/description/parli
st
/site/categories/category/name/(preceding-sibling::description
    | self::description | following-sibling::description)/parlist
```

6. Unions

   N1/(RLP1 | RLP2) → (N1/RLP1 | N1/RLP2)

   This rewrite rule changes a union at one step in a path expression to a union of two longer path expressions.

   Sample query pair:

```
/site/categories/(preceding-sibling::people
    | self::people | following- sibling::people)/
      child::*/watches/watch/@open_auction
/site/(categories/preceding-sibling::people | categories/self::people
    | categories/following-sibling::people)/
      child::*/watches/watch/@open_auction
```

Predicate Equivalences inside predicates

1. Built-in disjunctions

   [descendant-or-self::N] → [self::N or descendant::N]

   This rewrite rule is designed to check whether the descendant-or-self axis, which has a built-in disjunction, is handled the same way as the boolean disjunction operator inside a predicate. Of course, a similar rewrite rule can be written for ancestor-or-self.

   Sample query pair:

```
/site/categories/category/child::*[descendant-or-
self::parlist]
/site/categories/category/child::*[self::parlist or descendant::parlist]
```

2. Conjunctions

   [P1] → [P1 and P2], if P1 $\sqsubseteq$ P2

   This type of rewrite rule is based on [1]. It can occur that the right branch of a conjunction in a predicate is redundant, if any XML document satisfying the left branch also satisfies the right. [1] provides an algorithm for pruning such redundant nodes.

Sample query pair:

```
/site/people/person[watches/child::*/@open_auction]
/site/people/person[watches//@open_auction]
```

3. Predicate flattening

   N[P1[P2]] → N[P1/P2]

   This rewrite rule is based on [5], which uses the term “qualifier flattening”. The rule converts a nested series of predicates to a path inside a predicate.

   Sample query pair:

   ```
   /site/people/person/watches[watch[@open_auction]]
   /site/people/person/watches[watch/@open_auction]
   ```

4. Node test inside predicates

   AX::N → AX::*[self::N]

   This rewrite rule moves the node test from the path into the predicate. We do not predict that this would cause any problem for the engines, but provide it as a sanity check.

   Sample query pair:

   ```
   /site/people/person/watches/watch
   /site/people/person/watches/child::*[self::watch]
   ```

5. Predicate with independent truth conditions This equivalence is not best expressed with rewrite rules, and was not used in the performance of our benchmark because it did not fit the format of our evaluation. However, we mention it for completeness. If one has a document set and a query of the following form:

   ```
   a[/b/c]
   ```

   then /b/c only needs to be evaluated once per document in the document set. This query could be paired with a predicate with dependent truth conditions to check the relative speed.

Boolean Equivalences Tautologies and contradictions

1. Tautologies

   N → N:[E or not(E)]

   Any predicate containing a disjunction of an expression E and the negation of E is a tautology.

   Sample query pair:

   ```
   /site/people/person/watches/watch
   /site/people/person/watches/watch[@open_auction or not(@open_auction)]
   ```

2. Contradictions

   N:[E and not(E)] → /parent::*

   Any predicate containing a conjunction of an expression E and the negation of E is a contradiction. The XPath query engine should stop immediately upon recognizing the presence of such a contradiction. Note that the standard representation of a contradiction in

XPath, namely a reverse step (parent, ancestor, preceding) at the root, is itself a type of query that the engine should recognize as a contradiction, as discussed by [5].

The above classification is of course neither complete nor without ambiguities. Some query types may be assigned to more than one category. In addition, the rules may feed or bleed one another, since some rules introduce subterms to which rules from other categories may apply.

3.3.2 Manually Constructed DTD/Schema-Aware Queries

The main focus of our benchmark is on automatically generated queries. This is a natural beginning point for a benchmark of this sort, since it is just these sorts of regular equivalences that XPath query engines should have the easiest time addressing. However, we believe that expanding the query set into manually constructed queries that are aware of the DTD/schema, that is, the characteristics of the particular document class, may create more challenging tests for an XPath query engine. These queries can be much more domain-specific than the previous ones. A non-comprehensive sample of such queries is given here. This is an area where future expansion of this benchmark may begin.

DTD/schema-aware queries Sample equivalences making use of a DTD/schema

1. Loops

   child::N1/child::N2/child::N3 → child::N1/child::N2/following-sibling::N4/parent::N1/child::N2/child::N3, if the DTD/schema provides that N2 must have an N4 following-sibling

   This rewrite rule creates a loop in the middle of a path. From a node it goes to the sibling, then the parent, then back down to the node.

2. Needless traversals

   //N1 → //N2/ancestor::N1, if the DTD/schema provides that every N1 has an N2 descendant

   This rewrite rule creates a needless tep downward to N2, from a query that simply looks for N1's.

3. Counting

   child::N → child::N[last()], if the DTD/schema provides that there can be only one N child at the current point in the tree

   There are many possible redundancies having to do with the last() and count() functions which may appear when the DTD/schema is considered.

4. Unions with unsatisfiable terms

   LS1 → (LS1 | LS2 | LS3), where LS2 and LS3 are unsatisfiable given the DTD/schema

   Knowledge of the DTD/schema allows the addition of unsatisfiable terms; an engine which parses the DTD/schema should be able to remove such terms from the query during normalization.

4 Performing the Benchmark

We now put some of the things discussed above to work.

4.1 Environment

The benchmark was done on a Intel Pentium IV system clocked at 2.8 GHz with 512 MB of main memory. The operating system is Linux, kernel 2.6.10, with glibc 2.3.2 and a bogomips[2] count of 5538, which corresponds to the bogomips of other similar systems . We tested Galax in version 0.4.0 without the Jungle Storage System and Saxon in its open source version 8.3b. Furthermore, we used an existing wrapper script written in Perl to query the engines, and the XMark document generator v0.96.

4.2 Generation of Input

As indicated in previous sections we decided against automatically generating all of the input data. This is mainly due to the short time we had to develop and implement the benchmark. Furthermore, developing the automated process turned out to be more involved than we first expected.

The initial queries evolved from preliminary discussions throughout the implementation phase. Although far from being complete in any sense, the queries are intended to represent the general idea of our benchmark. That is, they are as comprehensive and subject-oriented as possible.

From the initial queries we created respectively seven and four equivalent queries. This was done in part by applying some rewrite rules or rewriting manually if we felt that an issue could not be addressed by a rewrite rule.

To create the XML documents to run the queries on we used the XMark document generator. For our benchmark we created five documents with scaling factor from 0.1 up to 0.5 with a constant increment. The size of the documents is 11 MB, 23 MB, 35 MB, 47 MB and 58 MB.

When running tests, the engines under discussion could not handle documents larger then 50 MB. Apparently, the engines also have a limit of around 10 MB for temporary internal storage which results in a heap overrun if violated. These constraints limit the scope of our benchmark.

From each group of equivalent XPath queries, for each XML document and for each XPath engine we made a experiment configuration to be fed into the wrapper. The wrapper calls the engines for the XML document and the XPath query and measures the performance. For each experiment configuration the wrapper creates a report file containing the recorded runtime values.

4.3 Interpreting the Results

The measured values give us information about the engines going far beyond the absolute values of speed. More interesting than each value o

Because we divided the equivalences into several categories the equivalent XPath queries are oriented towards those categories. In other words if one evaluates queries in one category one can measure how good an engine detects

[2] Perhaps best described as "the number of million times per second a processor can do absolutely nothing." See the bogomips mini-howto at `http://www.clifton.nl/`.

equivalences in this category. By systematically evaluating an engine against queries in all categories, the developer of this engine can detect weakpoints.

5 Results

Due to a strange behaviour of the testing environment we were not able to implement all the ideas from the previous sections. We ran two test sets - one for each of the first two categories of equivalences - on both Galax and Saxon.

5.1 Test set 1

The following queries are intended to exhibit different behaviour for queries differing by navigational equivalences.

1. `/site/people/child::*/watches/watch/@open_auction`
2. `/site/people/child::*/parent::*/child::*/`

   `watches/watch/@open_auction`
3. `/site/people/child::*/child::*/parent::*/`

   `watches/watch/@open_auction`
4. `/site/people/child::*/watches[child::watch]/`

   `watch/@open_auction`
5. `/site/categories/(preceding-sibling::people|`

   `self::people|following-sibling::people)/child::*/`

   `watches/watch/@open_auction`
6. `/site/(categories/preceding-sibling::people|`

   `categories/self::people|categories/following-sibling::people)/`

   `child::*/watches/watch/@open_auction`
7. `/site/(people[following-sibling::categories]|`

   `categories/self::people|`

   `categories/following-sibling::people)/child::*/`

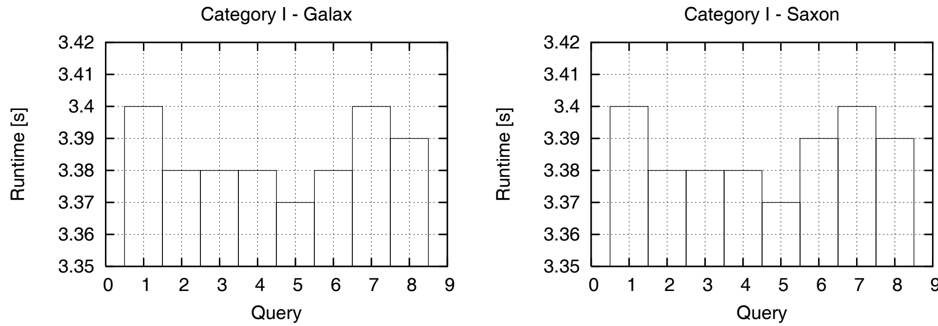
   `watches/watch/@open_auction`

8. /site/parent::*/child::site/people/child::*/

   watches/watch/@open_auction

The first and fifth query are initial. The second, third, and eight query have been derived from the first one by application of the rule "Redundant use of antagonistic axes". The fourth query has been generated from the first one by the rule "Redundant predicate". The sixth query has been generated from the fifth one by the rule "Unions". The seventh is equivalent, but has been created by hand. A sample is provided in the section 3.3.2 in the item "Unions with unsatisfiable terms".

We obtained run-time values for the corresponding queries as given in the figures below. The values do not differ enough to draw conclusions about the weaknesses of the engines.



There are two readily apparent explanations for the small variance in results. On one hand we obtained the queries merely by single application of one or two rewrite rules. Multiple application of those rewrite rules may point the differences out more clearly. Furthermore, the chosen rewrite rules are naïve such that they may probably be filtered out by the normalization stage of processing.

5.2 Test set 2

For the second test set we created all of the following queries manually. Note that none of those queries is satisfiable. However, they have in common that without preprocessing the queries against the DTD it is expensive to detect unsatisfiability.

1. /site/child::people/ancestor-or-self::watches/watch
2. /site/child::people/(ancestor::watches/watch or self::watches/watch)
3. /site/(child::people/ancestor-or-self::watches/watch and

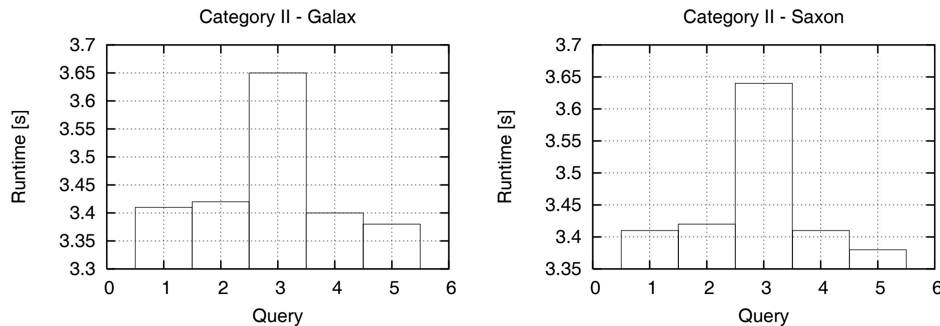
   ancestor::watches/watch)

4. [descendant::people[parent::site]])

5. /site[people[ancestor-or-self::watches/watch]]

6. /site/child::*[self::people]/ancestor-or-self::watches/watch

It is worth pointing out that the third query is obviously more difficult to process than the other queries. An explanation may be the two occurences of expensive ancestor calls. However, the measured differences between those queries do not meet our expectations.



6 Conclusion

In this report we gave some suggestions on how to properly set up a benchmark for XPath equivalent queries. Due to the preliminary nature of our study, we did not implement all of these suggestions. Both the collection of queries and the automatic generation procedure should be extended in future work.

As we have argued in section 3, our method of producing equivalent XPath queries might be only one step ahead of the normalization procedures in the engines. Further work might identify rewrite rules that cannot be efficiently included in the engines, although it is unclear whether identifying such rules would be of use to developers.

Our rewrite rules involve general equivalence only. However, it seems to be useful to consider equivalence w.r.t. a DTD or XSchema. Most documents with a priori knowledge available in the form of a DTD/XSchema offer chances for simplification of the corresponding rules.

During the evaluation procedure the engines turned out to be of little use when querying documents larger than 50 MB. For benchmarking XPath engines with storage systems like the relational interface Jungle for Galax and the Berkeley DB, the kind of queries or rewrite rules should be adapted, the same for their weighting in the final report.

In addition, implementation-specific details of the XPath engines should be taken into account in order to improve our benchmark. To keep our benchmark as objective as possible all engines on the market should be engaged, adding

Jaxen, CLaRK, XSQ, XML Spy, XHive, VAMANA, MS .NET, and JXPath as well as Saxon and Galax.

Furthermore, application-specific XPath engines, for example, those for mobile devices or XML streaming, require adapted rewrite rules or queries and shifted weightings for them. Streaming applications should be tested on how efficiently they can convert backward into forward axes, while mobile XPath engines [11] are unlikely to process documents larger than 2 MB and are rather restricted in memory.

By the same argument it may be wise to test for memory usage as well. In our benchmark we only considered runtime values. A finer-grained measurement could give information more valuable for the vendors of XPath engines.

References

[1] Amer-Yahia, Sihem, SungRan Cho, Laks V.S. Lakshmanana, and D. Srivastava. 2001. Minimization of tree pattern queries. Proceedings of ACM SIGMOD, May 2001, Santa Barbara, California, USA. Available at: `http://www.research.att.com/~sihem/publications/SIGMOD01.pdf`.

[2] Armstrong, Eric. Working with XML: The Java API for Xml Processing (JAXP) Tutorial. Available at: `http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/`.

[3] Flynn, Peter. The XML FAQ v4.0 (2005-01-01). Available at: `http://www.ucc.ie/xml/`.

[4] J. Gray. 1993. Quoted in [8].

[5] Olteanu, Dan, Holger Meuss, Tim Furche, and François Bry. 2001. Symmetry in XPath. Technical Report, Computer Science Institute, Munich, Germany. Available at: `http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2001-16.pdf`.

[6] Olteanu, Dan, Holger Meuss, Tim Furche and François Bry. 2002. XPath: Looking Forward. Proceedings of the Workshop on XML-Based Data Management (XMLDM) at EDBT, Prague, March 2002. Available at: `http://www.pms.ifi.lmu.de/mitarbeiter/olteanu/publications.html`.

[7] Oracle9i XML Database Developer's Guide - Oracle XML DB. Release 2 (9.2). Part Number A96620-01.

[8] K. Runapongsa, J.M. Patel, H.V. Jagadish, Y. Chen and S. Al-Khalifa. 2004. The Michigan Benchmark: Towards XML Query Performance Diagnostics. See `http://www.eecs.umich.edu/db/mbench/`.

[9] Schmidt, Albrecht, Florian Waas, Martin Kersten, Michael Carey, Ioana Manolescu and Ralph Busse. 2002. XMark: a benchmark for XML Data Management. Proceedings of the International Conference on Very Large Databases (VLDB), Hong Kong, China. Available at: Http://monetdb.cwi.nl/xml/index.html.

[10] XML Path Language (XPath). Version 1.0. W3C Recommendation, 16 November 1999. `http://www.w3.org/TR/xpath`.

[11] XML.com: Implementing XPath for Wireless Devices - Bilal Siddiqui. http://www.xml.com/pub/a/2002/06/05/wirelessxpath1.html